Knowledge Representation & Symbolic Reasoning Project (Group 8)

Bram Benist (4281926), Timo van Frankenhuijzen (4483448), Wouter Meijer (4391659)

***Abstract*— In this paper we present an investigation in building the architecture for a reasoning robot. ROSPlan, PDDL and Grakn will be used to make a robot perform shelf stocking duties in the AIRLab simulation. PDDL is used as an action knowledge base and planning system. Grakn is used for a product knowledge base and goal inference. ROSplan and python are used to connect the systems and simulate the solution. With the combination of these systems, an efficient solution is found for stocking various kinds of products. The unique architecture allows the system to be scalable to different kind of environments.**

.

I. Introduction

In this report, a stocking robot is presented in a small store simulation scenario. The goal of the simulation is to stock products from a loading table to their corresponding shelves while fulfilling packaging conditions.

In the simulation we make use of three product groups. The product groups are:

- Regular products
- Freezer products
- Fresh products

All of these groups have their own set of rules and initial conditions. All products have an initial condition that they start on the loading table. Products of the group **Regular products** can be immediately stocked on a normal cabinet shelf. Products of the group **Freezer products** need to be stocked on a shelf in the freezer. Products of the group **Fresh products** are not packaged placed on the loading table and need to be packaged first. After this they can be stocked on a normal cabinet shelf.

We now have a simulation setting, object groups and a set of rules. The task of the robot will be to stock all objects.

However, there is a possibility that an unknown object is placed on the loading table. By simply defining to which group the product belongs the robot should know which actions it has to take. For example, when there is an object named "cola", we only have to add information about the product group this new object belongs. From the rules stated earlier the robot should automatically know where to place the product and if additional requirements are necessary.

To accommodate the robot with a sequence of actions it needs to know what the conditions and effect is of every action. For example, a fresh product does need to be packaged before it can be placed on a shelf. Therefore we require a planner which knows basic information about the actions and knows what the chronological order needs to be to take them and reach the desired goal. This to allow the robot to independently reach the goal while only making use of the knowledge base.

To summarize, we have three different groups of products (fresh, freezer, and regular) which need to be stocked on their corresponding shelves by a robot. The robot should be able to access something to infer the goal and action sequence necessary. It should also have the ability to infer information about new unseen products when minimal information is given about this product.

II. Modelling The Task

In this chapter we will describe the knowledge design aspect of our project, the reasoning design will be presented separately in the subsequent section. The goal of this distinction is twofold, first to make the components we develop more modular and secondly to make the design more straightforward.

A. Modelled actions

Here we will present the actions possible for the robot. In the simulation, the robot, also known as TIAGo, will execute the stocking actions based on the rules from the knowledge base as stated in section I. TIAGo will be placed in its initial condition and it will have to pick up products. For this, we make use of two loading tables where the to be stocked products are placed.

The first action the robot will do is **move** to the *loading table*. It will thus change its *location* from its current position in the world to its destination position in the world. At its destination it will **pick** a product. For this TIAGo will look around and make an octomap from the environment to make sure it does not collide with objects. Then, it will pick a product with its right gripper only if the right gripper is *free*. To pick, TIAGo must position its arm, open the gripper, move in the direction of the product and then close the gripper. After the **pick** action is complete TIAGo *holds* the product it picked. Now TIAGo will move its arm back to its initial state and will begin to **move** to the next destination. This destination should be obtained from the knowledge base based on the product it picked-up. TIAGo thus should know from the knowledge base which kind of product it holds. At the destination it places the product on a freezer or regular shelf, or on the packaging station. For this it uses a **place** action which also makes use of an octomap. It is similar to the pick action but in reverse as it moves the arms to the destination location and opens the gripper instead of closes. After the **place** action it does not *hold* the product and has a *free* right gripper again. One special case is applied that

when an object is placed on the *packaging station* a product of the fresh products group will be packaged. Such a product needs to be packaged before being stocked on the shelf.

With these actions, TIAGo should be able to complete a goal formulated within the boundaries of the simulation. Here we have three actions: **move**, **pick** and **place**. These actions are written in first order logic in Equation 1.

$$\begin{aligned} &robot(TIAGo) \wedge location(TIAGo, x) \wedge move(TIAGo, x, y) \\ &\quad \implies location(TIAGo, y) \wedge \neg location(TIAGo, x) \\ \\ &robot(TIAGo) \wedge product(x) \wedge location(TIAGo, y) \\ &\wedge location(x, y) \wedge pick(TIAGo, x) \wedge free(TIAGo) \\ &\implies holds(TIAGo, x) \wedge \neg location(x, y) \wedge \neg free(TIAGo) \\ \\ &robot(TIAGo) \wedge product(x) \wedge holds(TIAGo, x) \\ &\wedge location(TIAGo, y) \wedge place(TIAGo, y) \\ &\implies location(x, y) \wedge \neg holds(TIAGo, x) \wedge free(TIAGo) \\ \\ &product(x) \wedge packaging\_station(y) \wedge location(x, y) \\ &\implies packaged\_at(x, y) \end{aligned} \tag{1}$$

B. Store Rules

In this subsection, we will formulate all the store rules in the form of a knowledge base in first order logic. Note that in the knowledge base only a specific set of products is listed. More products can be added later.

As mentioned in the introduction we have certain rules for products which allow us to determine the goal of a product. For example, hagelslag (chocalate sprinkles) is part of the regular product group, kroket is part of the freezer product group and bread is part of the fresh product group. In Equation 2 these rules are written in first order logic. l.5

$$\begin{aligned} &product(x) \wedge regular(x) \iff goal(location(x, y)) \\ &\wedge regular\_shelf(y) \\ \\ &product(x) \wedge freezer(x) \iff goal(location(x, y)) \\ &\wedge freezer\_shelf(y) \\ \\ &product(x) \wedge fresh(x) \iff goal(location(x, y)) \\ &\wedge regular\_shelf(y) \wedge goal(packaged\_at(x, z)) \\ &\wedge packaging\_station(z) \end{aligned} \tag{2}$$

Within the store rules we also want to be able to add new unseen products. So our knowledge base should have the ability to be able to have an instance of a product added. When a new product is added the user is asked to which group this product belongs. With only this small amount of information the knowledge base should be able to infer its corresponding goals.

C. World Model

For the world model, the AIRLAB supermarket simulation is used. In Figure 1, a map of the world is showed.

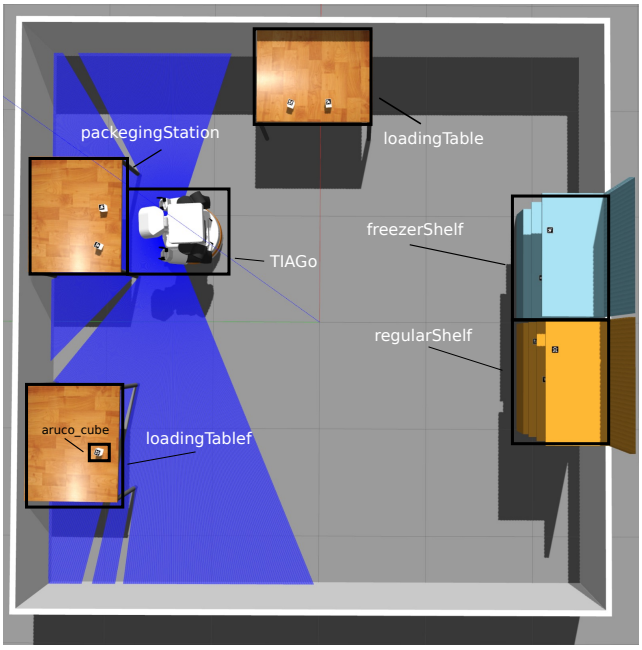

Fig. 1: Map of the environment

In the world model we make use of ”aruco_cubes_[three digit number]” to simulate products. We have three tables, simulating the two loading tables and the packaging station. There are two cabinets simulating the freezer and regular shelves.

D. Knowledge base Design Choice

In this section we will briefly explain our decisions and the reason why our final ontology is a good fit to the task at hand. A very pragmatic design approach was chosen with the mindset of: ”a good ontology is an efficient and representation of the information required for the mission at hand”.

We used Protege for our initial Ontology design.

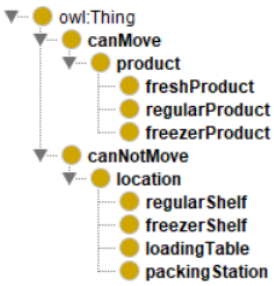

Fig. 2: Our initial ontology design in Protege

So the requirements from our initial design were that we need a knowledge base which can contain:

- knowledge about actions. So their pre and post conditions

- knowledge about how each product group should be handled
- knowledge about what the destination is for products from each product group

Given these requirements we considered combinations of the following options: PDDL, ROSplan, ROSProlog, Prolog(pyswip, pylog), Grakn and pure Python.

We have chosen for PDDL 2.1 with ROSsplan for the planning of the task and the knowledge base for the actions. PDDL allows a clear knowledge base for actions with conditions and effects. It also has to ability to plan with durations and optimize objective functions.[3] By creating an objective function to prioritize the completion of tasks and goals, PDDL will generate a plan where the robot will do tasks while waiting for the packaging of objects to complete. This increases efficiency of the robot and is a large advantage compared to Prolog or python for the planning of an action sequence.

The reason that we do not use PDDL for the product knowledge base and goal inference is that PDDL files are static. The only way to add a new unseen product is to manually specify its goals and conditions. It is hard to use the existing knowledge to infer info about new knowledge. To incorporate these benefits of a compounding knowledge base we have chosen to employ Grakn [1]. Grakn allows us to create a class hierarchy similar to OWL. The difference is that Grakn comes with tools to visualize the ontology database in real time.

```
# PRODUCTS
$prod-1 isa freezer_product, has name "kroket1";
$prod-2 isa regular_product,
    has name "hagelslag1";
$prod-3 isa fresh_product, has name "brood1";

# LOCATIONS
$loc-1 isa loading_table,
    has name "loading_table1";
$loc-2 isa packing_station,
    has name "packing_station1";
$loc-3 isa freezer,
    has name "freezer1";
$loc-4 isa shelf,
    has name "shelf1";
```

Fig. 3: Part of the knowledge base data in Grakn Query Language

To modify this hierarchy the Grakn query language is used instead of prolog. This language is more modern and intuitive compared to the logic programming language of prolog.

The figure above results from the data in Figure 3. The instances of each product are displayed and it is easy to see which instances share which attributes. The graph also shows a located-at relationship between each product and the loading table. This relationship was an experiment and is not utilized in our code. The other locations in the room are also visible at the bottom.

Grakns rules will be described in the next section, they make the database easily expandable with new incomplete information, it's modularity make it straightforward to add

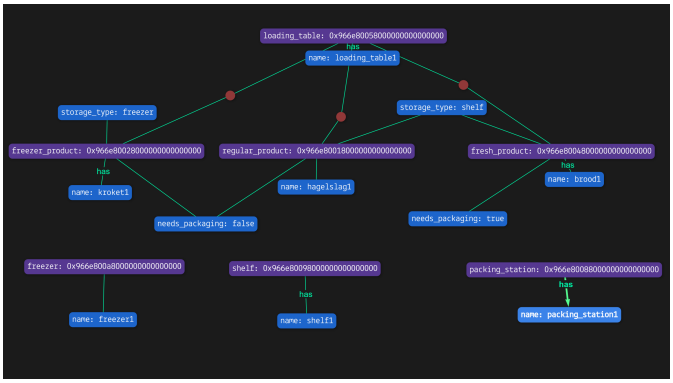

Fig. 4: The Grakn knowledge graph. The purple blocks are instances. The red dots are relations and the blue blocks are attributes

new product groups. Lastly, because the Grakn server has a clear and fast Python API we can access its contents quickly and use this to generate updated problem PDDL files rapidly.

To summarize, we use PDDL in combination with ROSplan for the planning of the actions required to achieve the goal. To make this possible it requires a knowledge base with all the actions. We use a Grakn knowledge base for the products and store rules. Grakn will reason from the knowledge base the required goals from the initial conditions in the world. This all is connected with ROS and python.

III. Reasoning and Planning

In this chapter we will present the reasoning and planning aspect of our project.

A. Grakn Reasoning

An overview of the Grakn components is given in subsection VII-B.

Grakn reasoning takes the form of inference based on rules. The rules are implemented as horn clauses under the hood. [2].

$$q1 \wedge q2 \wedge ... \wedge qn \rightarrow p$$

An example of the third store rule Equation 2 translated to a Grakn rule is:

```
rule fresh_products_need_packaging:
  when {
    $prod isa fresh_product;
  } then {
    $prod has needs_packaging true;
  };
```

Fig. 5: A Grakn rule which specifies that all fresh products need packaging.

Each rule can have only one consequence, so we need 2 rules with the same condition and different consequences to fully encapsulate the first rule.

The inference takes place at query time. That means that when we send our database a query which it cannot answer with the explicit data it contains, it then looks whether it

can infer the answer using the rules in its schema. This is implemented by allowing the user to add any new product at the start just by specifying its product group and name. Our system is then able to infer where the product needs to be placed and whether it requires packaging, this is useful because it allows drawing wide conclusions on narrow observations.

B. PDDL Reasoning

The PDDL domain file has the knowledge base about the actions the robot can perform. These actions were described in subsection II-A. Additionally the predicates: object-at, packaged-at, robot-at, is_holding, free and can_move are defined. We have the types: waypoint, robot, object and gripper. With these actions, types, and predicates we are able to obtain a plan via ROSPlan that obtains the goal. When a goal is defined with packaged-at(product location) it will place the product on a table after which it will pick it up again.

All of the actions, predicates and types can be connected to atomic terms as stated in the first order logic functions in section II. A few exceptions are present though. The type waypoint is a location which can be used in the atom location(x,waypoint) when stating the location of a robot or an object. Since the robot has two grippers we have the type gripper as a separate object from the robot. This allows the robot to hold two objects, though this has not been implemented. Free also corresponds to a gripper instead of the robot. Lastly, we have the *can_move* predicate which increased efficiency by moving directly to a location. The solver we used sometimes planned two subsequent move actionsby this predicate since it is condition for the move action. It turns off after a move action and turns on after a place or pick action.

We make use of *Partial Order Planning Forwards* (POPF) planner interface with ROSplan for planning. This planner comes as default with the singularity image of ROS for the course. It supports the use of *durative-actions* and *numeric-fluents* in PDDL 2.1. The actions are defined as durative-actions since they take a time interval to succeed. The numeric-fluents allows us to create an objective function which can have a goal to be maximized or minimized.

We have also implemented PDDL domain files which had an extra action called 'packaging'. This allowed us to simulate packaging for a certain duration. By also implementing objective functions to maximize within the goal of PDDL. By doing this the planner should come with a solution where the robot will do other actions while waiting for the packaging to finish. However, we were unable to simulate this due to failing picking actions. How this failed is further discussed in the results.

IV. Results

In this section we will present the results of our simulation. The simulation contains a few parts.

A. Knowledge Base

The knowledge Base consists of multiple elements. One part is described in the PDDL file. This file includes all knowledge about the actions and inferred knowledge about the initial conditions. The other part of our knowledge base is in Grakn. We succeeded to make a knowledge base with Grakn which implements inference. This means that we can infer essential mission parameters from incomplete user input. Also, our system in Grakn is simple to expand and easy in use. The inference allows us to add a product instance which only requires information about the product group.

B. Reasoning and Planning

First TIAGo must reason which product has which goal location. This is done using Grakn rules. After TIAGo has obtained the goal location, it will add it in the PDDL file. Thereafter, TIAGo needs to plan a sequence of actions to achieve its goal. This planning is done by the reasoner based on the PPDL files.

Also, the planner is able to plan the packaging time at the packing station (see Figure 6 in the Appendix). This allowed the robot to not wait for the packaging of the product and do other tasks. This increases the efficiency of our robot depending on the time it takes to package an item and the ratio of fresh products and other products. Below you see a table for a system where packaging is 6 seconds and every other action is 2 seconds

non-fresh product	2	1	1	1	2	3
fresh product	3	3	2	1	1	1
time-saved (s)	18	10	10	6	6	6

TABLE I: time saved based on non-fresh and fresh products to be stocked

From the the table we can see that the time-saved is an equation in the form of time saved = $min(5t_a \cdot NF, t_p \cdot F)$. Here NF = non-fresh products, F = fresh products, t_a = time of an action, and t_p = time of packaging.

However due to time limitations, we were not able to implement this in the final simulation. Though we have added the problem and domain files in the appendix subsection VII-C

C. Execution

For the execution we made use of the provided ROS action pick and move and built a placing action described in subsection II-A. TIAGO is able to pick and place the desired product. However, unfortunately the underlying simulation is not working consistently. We had to endure many random errors about getting stuck and not finding a path in one run, whilst running without issue in previous and subsequent runs.

To conclude, in the simulation, TIAGo is able to stock the given products on their corresponding shelves successfully. Moreover, in the PDDL, a plan can be made where TIAGo keeps stocking other products while waiting for the fresh product to be packaged.

The link below leads to a video showing our robot perform a pick and place action for a new unseen product, we only specified its product group.

https://youtu.be/-avgNHraw10

V. Discussion

The goals of stocking different products according to their store rules is achieved. By only requiring information about the group of which the product is part of, the goal of the product can be inferred from the knowledge base. We unfortunately did not finish a working simulation with the optimal planning.

Though most of our goals are achieved, still several limitation can be identified. Our system runs its inferences once at the start and then generates a PDDL file. So it is reactive with regards to initial conditions. Although if something changes in the simulation, there is no possibility to adapt the plan which has been formulated.

Another limitation of Grakn is that rules can only have one consequence, this limitations caused convoluted code.

The last limitation of our work is the lack of measurements for comparison. By having unstable simulations and to much time spending on fixing simulation issues we were unable to give proper results to compare our system to others.

VI. Future Work

It would be good for the reactivity if when a discrepancy between the KB and the measurements is witnessed we could trigger an update of the KB and a rerun of the entire planner. This would make the robot reactive to changes at runtime by allowing dynamic replanning and inferences based on new information.

Another interesting aspect of Grakn to investigate would be the relationship functionality. Currently the storage types are inferred as strings, it is possible to create instances for each storageShelf and specify a goal location as a destination-at relationship between the storageShelf. This is construct probably becomes more interesting for larger knowledgbases or very dynamic knowledgebases.

References

[1] *Grakn vs OWL*. 2020. URL: `https://brightspace.tudelft.nl/d2l/le/content/318968/viewContent/2094996/View`.

[2] *Rules — GRAKN.AI*. `http://docs.grakn.ai/docs/schema/rules`. (Accessed on 04/06/2021).

[3] *wiki of PDDL*. URL: `https://planning.wiki/`.

VII. Appendix

A. README overview

The focus of the demo is on showcasing our Grakn-PDDL files which can dynamically generate PDDL files for new products and varying configurations of existing products based on limited user input.

We wrote 2 shell script to automate running the interactive and static demo of our system. It is essential to specify the location of the Singularity image in the corresponding shell script. For further running please see our in-depth README in the repository.

B. Grakn overview

Grakn has 3 components. The Grakn server, the Grakn console and the Grakn-client. The server runs in the background and contains the database. The console is used to manage the database. the Grakn-client is a python module which allows us to interface Python with the Grakn server. We interact with the database by sending READ and WRITE queries. The language for these queries is the Grakn query language written in `gql` files.

The Grakn framework enforces a separation between the structure of your data and the data itself. You describe the structure of the data in the schema using the `define` keyword and you describe the data itself in the same gql language using the `insert` keyword.

C. PDDL files with object function

1) Domain file:

```
(define (domain pick_place)

    (:requirements :typing :durative-actions :fluents
        )

    (:types
        waypoint
        robot
        object
        gripper
    )

    (:predicates
        (active ?obj - object)
        (requires-packaging ?obj - object)
        (packaged-at ?obj - object ?wp - waypoint)
        (robot-at ?v - robot ?wp - waypoint)
        (object-at ?obj - object ?wp - waypoint)
        (is_holding ?g - gripper ?obj - object)
        (free ?g - gripper)
        (can_move ?v - robot)
    )
    (:functions
        (efficiency)
        (object_function)
    )

    (:durative-action move
        :parameters (?v - robot ?from ?to - waypoint)
        :duration ( = ?duration 2)
        :condition (and
            (at start (robot-at ?v ?from))
            (at start (can_move ?v))
        )
        :effect (and
            (at end(increase (object_function) (efficiency)))
            (at end (robot-at ?v ?to))
            (at start(not(robot-at ?v ?from)))
            (at end(not(can_move ?v)))
        )
    )

    (:durative-action pick
        :parameters (?v - robot ?wp - waypoint ?obj - object ?g - gripper)
        :duration (= ?duration 2)
        :condition (and
            (over all(free ?g))
            (at start(object-at ?obj ?wp))
            (over all(robot-at ?v ?wp))
            (at start(active ?obj))
        )
        :effect (and
            (at end(increase (object_function) (efficiency)))
            (at end (not (object-at ?obj ?wp)))
```

```
            (at end (not (free ?g)))
            (at end (is_holding ?g ?obj))
            (at end (can_move ?v))
        )
    )

    (:durative-action place
        :parameters (?v - robot ?wp - waypoint ?obj - object ?g - gripper)
        :duration (= ?duration 2)
        :condition (and
            (at start(is_holding ?g ?obj))
            (over all(robot-at ?v ?wp))
            (over all(>=(object_function) 2))
        )
        :effect (and
            (at end (not(active ?obj)))
            (at end(increase (object_function) (efficiency)))
            (at end (free ?g))
            (at end (not(is_holding ?g ?obj)))
            (at end (object-at ?obj ?wp))
            (at end (can_move ?v))
        )
    )
    (:durative-action packaging
        :parameters (?obj - object ?wp - waypoint )
        :duration (= ?duration 6)
        :condition (and
            (at start(requires-packaging ?obj))
            (over all(object-at ?obj ?wp))
        )
        :effect (and
            (at end (active ?obj))
            (at start(assign (object_function) 0))
            (at start(assign (efficiency) 1))
            (at end(assign (efficiency) 0))
            (at end (packaged-at ?obj ?wp))
            (at end (not(requires-packaging ?obj)))
        )
    )

)
```

2) Problem file:

```
(define (problem pick_place)
    (:domain pick_place)
    (:requirements :strips :typing)


    (:objects   tiago - robot
                rightgrip - gripper
                        wp_table_2 - waypoint
                        wp_table_1 - waypoint
                        wp_table_3 - waypoint
                wp_cabinet_1 - waypoint
                wp_cabinet_2 - waypoint
                        wp0 - waypoint
                aruco_cube_582 - object
```

```
                aruco_cube_444 - object
    )
    (:init
        (=(efficiency)0)
        (=(object_function)2)
            (robot-at tiago wp0)
        (object-at aruco_cube_444 wp_table_3)
        (object-at aruco_cube_582 wp_table_1)
        (requires-packaging aruco_cube_444)
        (active aruco_cube_582)
        (active aruco_cube_444)
        (free rightgrip)
        (can_move tiago)

    )

    (:goal (and
    (packaged-at aruco_cube_444 wp_table_2)
        (object-at aruco_cube_582 wp_cabinet_1)
    (object-at aruco_cube_444 wp_cabinet_2)
        ))

        (:metric maximize (object_function))
)
```

D. PPDL plan

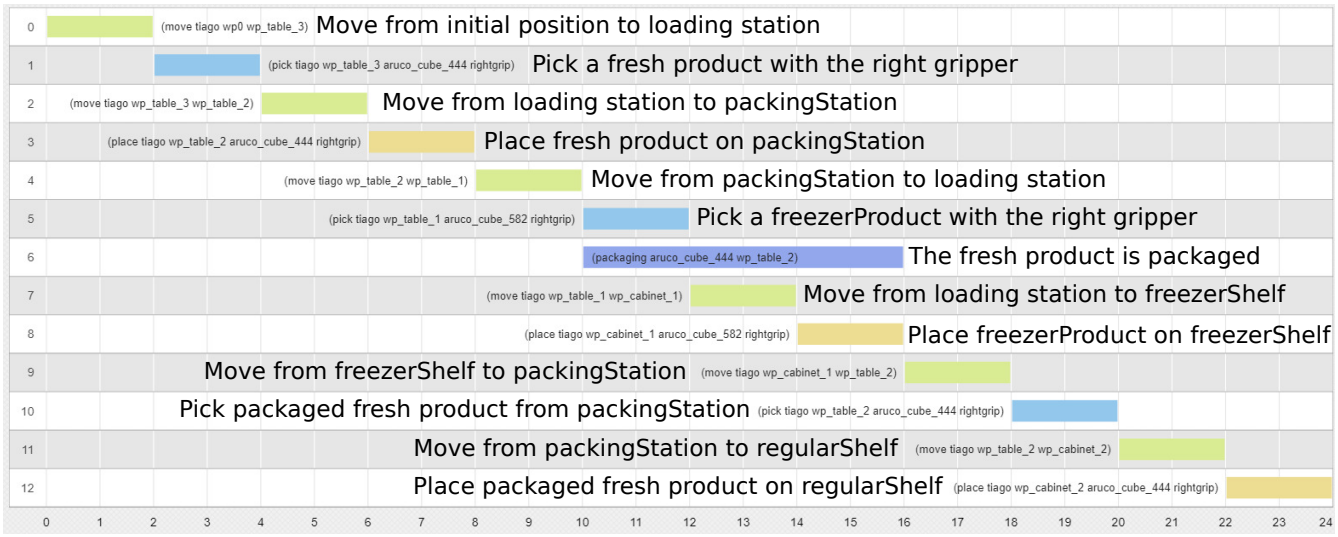

Fig. 6: PDDL plan