Automated Planning Theory and Practice Project Report

Ziglio Riccardo

University of Trento
Master's Degree in Artificial Intelligence Systems
riccardo.ziglio@studenti.unitn.it

July 2022

1 Introduction

This report details the proposed solution for the assignment on PDDL and Plansys2 for the Automated Planning Theory and Practice course [14]. The assignment is structured in 4 sub-problems, each building on the previous one. In this introductory section the organization of the delivered zip archive and of the rest of the document will be described.

1.1 Archive organization

The archive is composed by 4 folders, named after each problem, containing all the necessary files for solving the problem. In particular each folder consists of:

- a **src** folder, containing the **domain.pddl** file and the **problem.pddl** file,
- a **command.txt** file, containing the commands used to run the planner,
- an **output_plan.txt** file, containing the generated plan.

Only for the **problem4/plansys2_assignment** folder, this structure is not respected since it contains the solution for the Plansys2 part of the assignment.

1.2 Report structure

The document is structured as follows for each problem:

1. Understanding of the problem - section that describes which assumptions were made to model the problem
2. Predicates descriptions

3. Actions description

4. Goal description

2 Problem 1

2.1 Problem Understanding

The first problem defines the scenario on which the other problems will be built, the *injured-people-scenario*. The domain file contains three actions: **load**, **unload**, and **move**. The domain file also contains two constants: the **depot**, since it is present in all successive problem instances it could be considered a constant, and the robotic **agent**, since the assignment states *"[...] we use a separate type for robotic agents, which currently happens to have a single member in all problem instances"*. Thus, a separate type for for robotic agents, (*robot*), has been defined. Also, since *"Crate contents shall be modeled in a generic way, so that new contents can easily be introduced in the problem instance"*, the **crate** type has been defined. Figure 1 represents the domain hierarchy for the defined types.
Note that even if theoretically each type should be a descendant of the most general type *object*, due to some misinterpretation by PDDL, it was necessary to define this dependency explicitly.

The *LAMA* planner was used to generate a solution for this part of the assignment.

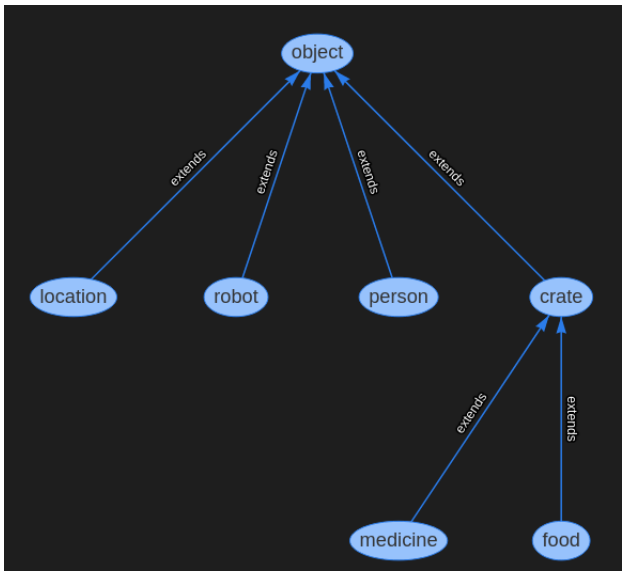

Figure 1: Problem 1 Domain Hierarchy.

2.2 Predicates Description

The predicates: **robot_at(?r - robot ?l - location)**, **crate_at(?c - crate ?l - location)**, and **person_at(?p - person ?l - location)** allows to define the exact position of entities in the scenario. The same result could have been achieved by using the predicate at, i.e. at(robot, location), but it seem more intuitive to use three different predicates to locate the different objects. The predicates **carrying(?r - robot ?c - crate)** and **free(?r - robot)** defines either if the robot is carrying a crate (as stated by the problem description, *"each robotic agent can pick up one single crate"*) or if the robotic agent is free, so it is not carrying any crate. The last predicate is the **has_crate(?p - person ?c - crate)** and defines if the person has a certain crate (i.e. of food or medicine), and will be useful in defining the goal.

2.3 Actions Description

Three actions were defined in the domain file: **load, unload** and **move**.

The **load** action allows the robotic agent to pick up a single crate, granted it is not already carrying a crate and both the robotic agent and the crate are at the same location (as described in the assignment). The **unload** action allows the robotic agent to deliver the crate to the injured person whose required it, to do so they both have to be in the same location. The **move** action allows the robot to move between a source (src) location and a destination (dst) location. These may also not be adjacent since *"The robotic agents can move directly between arbitrary locations"*. More in detail:

- **load**(?r - robot ?c - crate ?l - location)
  - *precondition*: crate and robotic agent must be at the same location; the robotic agent must not be already carrying a crate (since it can only pick up one).
  - *effect*: the robotic agent is carrying the crate (so is no more free).
- **unload**(?r - robot ?c - crate ?p - person ?l - location)
  - *precondition*: the robotic agent must be carrying the crate; the the robotic agent can deliver the crate to the injured person who needs it, only if they both are in the same location (as stated in the assignment).
  - *effect*: the person has received the desired crate and the robotic agent is now free (is not holding a crate anymore).
- **move**(?r - robot ?src ?dst - location)
  - *precondition*: the robotic agent is at the src (source) location; src and dst (destination) must be different locations in order for the robot to actually move.
  - *effect*: the robotic agent arrives at destination.

2.4 Goal Description

As stated by the assignment, *"the goal of the robotic agent is to deliver to the needing people some or all the crates located at the depot"*. But, *"since there can be more than one person at any given location and each person either has or does not have a crate with a specific content, it is not sufficient to track where the crate is in order to know which people have received which crates"*. Also, *"people don't care exactly which crate they get, so the goal should not be (for example) that Alice has crate5, merely that Alice has a crate of food"*. Therefore, to verify if a person has received the desired crate, or crates, its sufficient to check the **has_crate** predicate w.r.t. the available supply crates defined in the problem. The resulting goal formula is then a set of conjunction of disjunctions, defined using the *:disjunctive-preconditions* in PDDL.

Note that the same goal could also be defined using the *:existential-preconditions* (so using the exist logical operator instead of the or). In this case the goal definition would become something similar to: there exists a crate of a certain supply such that it is that person's desired supply crate; so the goal is for that person to acquire that kind of crate. This alternative goal definition, which may be more elegant than the proposed one, is reported as a comment in the problem file.

To develop a more realistic and efficient solution, given the scenario described in the assignment, the robotic agent is returned to the depot once all the crates have been delivered to the people in need. Without this additional goal, the robotic agent will stop at the last location it has delivered a crate to, once the plan has terminated. In a real emergency service logistic scenario (the one the assignment is inspired to), this would mean waiting for the robot to return back to the depot in order to satisfy the initial conditions (in particular *"initially all the crates are located at a single location that we may call the depot [...] a single robotic agent is located at the depot to deliver crates"*) for the execution of a new plan. This time spent waiting for the robot to return to its initial position may prevent the best solution to the problem from being found and, since the starting point is not ideal, that could lead the planner to generate a sub-optimal solution. Also, in a realistic emergency setting this time delay may bear significant consequences for the people needing supply crates.

3 Problem 2

3.1 Problem Understanding

In the second problem a carrier has been added to the robotic agent, which is now able to carry up to four crates at the same time. Moreover, the assignment also states that *"the capacity of the carriers (same for all carriers) should be problem specific. Thus, it should be defined in the problem file"*. To model the carrier capacity two solutions have been proposed, one based on *numeric-fluents* and another using *math-predicates* (located respectively in the *problem2/fluents* and *problem2/predicates* folders). Both solutions share the underlying idea that

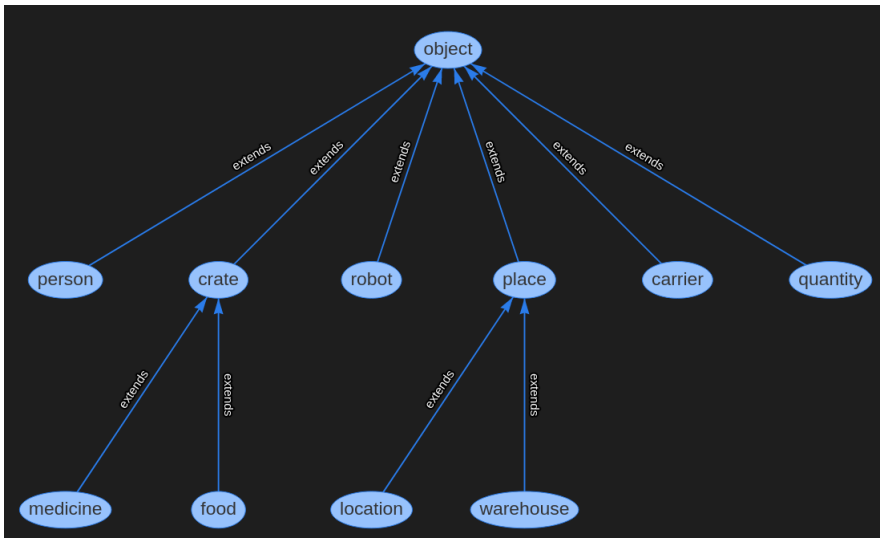

Figure 2: Domain Hierarchy Problem 2.

the robotic agent and carrier are attached forming a single unit, meaning that they both move together, and the robotic agent is the one in control of loading/unloading crates and moving the carrier between locations. While they both solve the same problem in slightly different ways, the *math-predicates* solution is compliant with the assignment request. In fact, by using predicates is possible to define the carrier capacity in the problem file. The same cannot be said for the *numeric-fluents* solution, where the capacity can only be defined (as an integer) in the domain file. Thus, making it inaccessible in the problem file as required by the assignment. Despite this, this solution has been described as another possible resolution of the problem.

Figure 2 shows the domain hierarchy for the *math-predicates* solution, highlighting the different types introduced:

- **quantity**: used to model the carrier capacity;
- **carrier**: introduced to meet the problem requirements since *"[...] there should still be a separate type for carriers"*;
- **place**: used to differentiate between places where injured people are (i.e. the **location** type) and places where creates are stored (i.e. the **warehouse** type).

The *Fast Downward* planner [6] has been used to generate a solution for the *math-predicates* problem, while the *ENHSP-2020* [2] planner was used for solving the *numeric-fluents* version.

3.2 Predicate Description

Some additional predicates have been introduced for the *math-predicates* and *numeric-fluents* solutions w.r.t. the ones already presented in 2.2 and will be presented in the next sections.

3.2.1 Math-Predicates solution

In order to model the carrier capacity the **quantity** type was introduced. Also, the **num_crates(?k - carrier ?q - quantity)** predicate is used to keep track of the number of crates loaded onto a carrier. To manage the loading and unloading of crates, so the increase or decrease of the number of crates carried, the **inc(n1 n2 - quantity)** and **dec(n1 n2 - quantity)** predicates were introduced. These model the mathematical relation of the increase and decrease of the number of crates transported by the carrier in the following manner (note that increase and decrease are reserved keywords for the solution based on fluents, therefore cannot be used to model these predicates):

- **inc(q, q1)**: $q0 < q1 < q2 < q3 < q4$
- **dec(q1, q)**: $q0 > q1 > q2 > q3 > q4$ (meaning that q0 is smaller than q1 and so on)

Using this definition different quantities can be describe in the problem file to model the number of crates loaded onto the carrier: i.e. **q0** refers to the empty carrier, **q1** indicates only one crate is loaded, **q2** two crates, and so on up to **q4** indicating maximum carrier capacity.

A new location predicate for the carrier has been added, **carrier_at(?k - carrier ?pl - place)**, since it is necessary for the robot and the carrier to be in the same position in order to load or unload crates as they are considered as a single unit (as explained in the previous section 3.1). Also, the **carrying(?r - robot ?c - crate)** predicate has been modified into **carrying(?k - carrier ?c - crate)** since now the robotic agent is equipped with a carrier. The **free(?r - robot)** predicate is no more used since it will actually block the execution of the **load** and **unload** actions. In fact, for the **load** action the robot should be **free** before the execution of the action and will become **not free** afterwards and will remain so in all successive action instances (the same for **unload**, which will keep the robot **free** in all successive states). Thus, making impossible for the robotic agent to load more than one crate at a time onto the carrier, in fact, it will need to load and then unload a crate to verify the **load** preconditions. Instead, the robot should be free at the beginning, not free when holding a crate, and then free again at the end of the action when the crate has been loaded (the opposite for the unload action). As it will be shown in the next section, this can be expressed using *:durative-action*, where using the conditions *at start(free robot)* and effects *at start(not free robot), at end (free robot)*, it will be possible to show the effect of the robotic agent taking a crate and loading it onto the carrier.

3.2.2 Numeric-Fluents solution

Fewer predicates were added in this solution compared to the previous one. The capacity of the carrier is now managed through the function **num_crates_carrier**, which keep track of how many crates the carrier is holding. In fact, *"A numeric fluent, similar to a predicate, is a variable which applies to zero or more objects and maintains a value throughout the duration of the plan"* [5]. Therefore, is no more necessary to define a type **quantity** as in the *math-predicates* solution. By using numeric fluent is possible to keep track in an "automatic way" of the number of crates loaded on the carrier. So, its sufficient to define the max number of crates (i.e. 4) the carrier can carry in the definition of the load/unload action. In particular, the action precondition should verify whether:

- **load**: **num_crates_carrier** < 4, meaning is possible to load a crate since the number of carried crates is smaller than the maximum capacity.
- **unload**: **num_crates_carrier** > 0, meaning is possible to unload a crate since the carrier is not empty.

Also, by using *numeric-fluents* the **num_crates_carrier** value can be modified using the *increase* or *decrease* keywords. As a consequence, the predicates to manage the increment and decrement between quantities are no more used, simplifying the problem definition.

As mentioned above, although the use of *numeric-fluents* does not allow to define the carrier capacity in the problem file, this solution has been reported for completeness.

3.3 Action Description

Both domains are composed by a total of four actions: **load**, **unload**, **move_to_deliver**, and **move_to_depot**.

- **load** (?r - robot, ?c - crate, ?k - carrier, ?wh - warehouse, ?startq ?endq - quantity)

  As described in 2.3, also in this case it is supposed that a robotic agent can only pick up/put down one crate at a time.

  - **preconditions**: the robotic agent, the crate and the carrier must all be at the same location; the carrier must not carry the desired crate already, otherwise it means it is not possible to load it; there must be an *increment* between the *starting_quantity* and the *end_quantity*, meaning we are loading more crates than the ones already carried by the agent; and also we must verify the carrier is actually starting from *start_quantity* number of crates (i.e. starting from **q0, q1, q2, ...**, meaning the carrier is empty, has only one crate, two crates, etc.).
  - **effects**: the crate is now loaded onto the carrier, so its no more at the initial position, and the carrier has increased the number of crates carried from *start_quantity* to *end_quantity*.

- **unload** (?r - robot, ?c - crate, ?k - carrier, ?p - person, ?l - location, ?startq ?endq - quantity)
  - **preconditions**: the robotic agent, the carrier and the person must all be at the same location in order for the crate to be delivered to the person requesting it; there must be a decrement between *start_quantity* and *end_quantity* (since we are unloading crates from the carrier); also in this case we must check that the carrier actually has loaded a *start_quantity* number of crates in order to execute this action.
  - **effects**: the person has now the create, which is no more loaded on the carrier, as a consequence the number of creates decreased from *start_quantity* to *end_quantity*. Now the robotic agent can either move to another location for a new delivery or to go back to the depot only if its empty, so it has delivered all the loaded crates.
- **move_to_deliver** (?r - robot, ?k - carrier, ?src - place, ?dst - location)
  - **preconditions**: the robotic agent and the carrier must be at the same location; the source and destination must be different locations (otherwise the agent cannot move), and the carrier must not be empty (otherwise it needs to go back to the depot to collect more crates).
  - effects: the robotic agent and the carrier are at the desired destination, ready for deliver crates.
- **move_to_depot** (?r - robot, ?k - carrier, ?src - place, ?dst - warehouse)
  - **preconditions**: the robotic agent and the carrier must be at the same starting location and the carrier must be empty.
  - **effects**: the robotic agent and the carrier will be at the depot an no more at the starting location.

From the description of the actions **move_to_depot** and **move_to_deliver**, is possible to understand why new type **place** was introduced. Especially for the first action, if the type **location** was used to describe the position of the agent (robotic agent and carrier) we would have incurred in some difficulties. In fact, without distinguishing between location and depot (or warehouse), the planner was not able to apply the correct **move_to_depot** action to reach the goal, but instead applied the **move_to_delivery** action, despite the carrier being empty at the last step of the plan. This undesired behavior was found in the *numeric-fluents* solution, while the *math-predicates* solution seem to not be affected by this. Consequently, the distinction between **location** (where the injured people are located) and **depot** (where the crates are located) through the **place** type ensures that the robotic agent reaches the correct destination (either a location or the depot) for both **move_to_deliver** and **move_to_depot** actions. In fact, in both cases the starting location could be anywhere on the map, so a generic **place**, and starting form there the agent

has to reach either a specific location (to where deliver crates) or the depot (to load more crates). Moreover, by using the **place** type the problem results more generic as its possible to expand the problem by adding new locations, i.e. more depots.

3.4 Goal Description

No difference was introduced w.r.t the previous goal definition in 2.4. Again an alternative goal formulation in terms of *:existential-precondition* is proposed within comments. Also for this solution the *additional goal* of the robotic agent returning to the depot once all the crates have been delivered is applied.

4 Problem 3

4.1 Problem Understanding

The third part of the assignment asked to convert what was done in the *Problem 2* to use *:durative-actions* [4]. The solution was implemented on top of the *math-predicates* solution presented in 3.2.1. This was preferred to the *numeric-fluents* solution since it allows to define the capacity of the carrier in the problem file as requested by the assignment.

For testing the robustness of the proposed solution, both a *single-agent* and a *multi-agent* case were developed (they can be found respectively in the *problem3/single-agent* and *problem3/single-agents* folders). The differences between the two are minimal, the domain files in the *src* folder are identical while the problem file for the *multi-agents* case is just a more complex version of the same problem, it just defines additional objects such as **carrier**, **robot**, **location** and **person**, but maintains the same goal and initial condition as the *single-agent* solution. Therefore, the remaining of this section will detail only the content of the *single-agent* case. A plan for both cases has been obtained through the application of two different temporal planners: **OPTIC** [3] and **Temporal Fast Downward** (TFD) [1] (the execution commands for both planners can be found in the *command.txt*), these two planners have been selected as they were studied during the course.

From the consultation of the **OPTIC**'s documentation some limitations of this planner have been identified, in particular, the missing support of: *existential-preconditions*,*disjunctive-preconditions* and *negative-preconditions*. Of course, this has imposed some changes in the implementation:

- the goal definition was reformulated, since expressed by means of disjunctions;
- complementary predicates were defined, such as **needs** and **does_not_need**, due to the impossibility of using *negative-preconditions* in the goal definition;

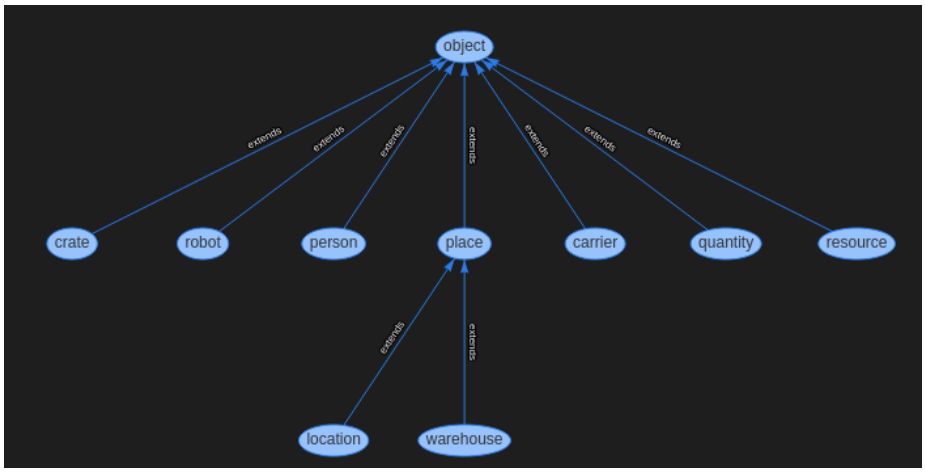

Figure 3: Domain Hierarchy Problem 3.

- a novel type **supply** was introduced to model the crate content, as shown in Figure 3

The plan parsed by **OPTIC** can be found in the *optic_output_plan.txt* file and in the *optic_output_plan_back_to_depot.txt* file (for both the *single-agent* and *multi-agents* solutions), the latter includes the additional goal, already mentioned in 2.1, of returning the robotic agent to the depot once all the crates have been delivered (done to achieve a more realistic modeling of the solution).

As stated above also the **TFD** planner was used to generate a plan (from the same problem definition parsed by the **OPTIC** planner). This was done both for completeness and to provide a comparison between the two planners introduced in the course. Also for **TFD** the parsed plan can be found in the *tfd_output_plan.txt* and in the *tfd_output_plan_back_to_depot.txt*. As studied during the course it is expected that **TFD** will find a more efficient solution (meaning a shorter plan) w.r.t. **OPTIC**, which still posses the feature of finding an optimal solution. Unfortunately, this assumption seems to not be verified since both plans obtained for the *single-agent* case and the *multi-agents* case are actually of similar length.

Moreover, both planners present small inaccuracies in the way they compute the plan. In particular, they either use only one agent (intended as a single unit composed by the robotic agent and the carrier) for the majority of the plan, or cannot have more than one agent in the same location at the same time (i.e. at the depot) and therefore one of them is forced to move (to deliver something) to another location even if empty. This behavior may be due to a wrong implementation of the agent behavior in the domain file or might be the case that the planner picks up some implicit constraints that alter the expected computation of the goal. However, both planners achieve a reasonable solution, in fact, all the crates are delivered to the people requesting them and the agent

returns at the depot once deliveries are completed.

4.2 Predicate Description

Most of the predicates remain unchanged w.r.t. what presented in 3, the novelties are reported below:

- the **free** predicate has been reintroduced to manage the movement of the carrier. The underlying idea is that the carrier can move to deliver a crate only in case the robotic agent is free, meaning is not holding any crates (so is not loading/unloading crates);
- **crate_content** defines the content of the crate based on the new type **supply**. This was a necessary introduction since **OPTIC** does not support *:disjunctive-precondition*;
- the predicates **needs** and **does_not_need** are used to replace the **has_crate** predicate (defined in the previous solution) and to model each person desired crate in the goal definition;
- the **has_carrier** predicate has been introduced only for the *multi-agents* solution, due to the wrong selection of the carrier by the robotic agent during the parsing of the plan.

4.3 Action Description

As previously stated both *single-agent* and *multi-agents* domain files are composed by four actions: **load, unload, move_to_depot, move_to_deliver**, presenting slight differences w.r.t. what presented in 3 due to the conversion of the problem to use *durative-actions*.

- **load**(?r - robot ?c - crate ?k - carrier ?wh - warehouse ?startq ?endq - quantity)
  - *duration*: 2 sec
  - *condition*
    - *at start*: the robotic agent, the carrier and the crates must all be at the same location. In the *multi-agents* case the robotic agent is linked to a specific carrier through the **has_carrier** predicate. There must be an increment from *startq* and *endq* since we are loading a crate. Lastly, the agent must be **free** to grab a new crate and load it onto the carrier.
    - *over all*: for all duration of the action the carrier has **startq** number of crates, meaning that no other crates can be loaded onto the carrier (from this agent or other agents, which will not be the ideal behavior since robotic agent and carrier are paired). As already stated in 3.1, also here a robotic agent can only load

one crate at a time onto the carrier as declared by the assignment: *"a robotic agent cannot pick up several crates at the same time"*.

  - *effect*
    - *at start*: the robotic agent is holding a crate.
    - *over all*: at the end of the action the crate will have been loaded onto the carrier; the crate is no more at the depot since it is now loaded on the carrier, and the number of crates has increased from **startq** to **endq** (i.e. from *q0* to *q1*, meaning one crate has been loaded). Once the crate has been loaded onto the carrier the robotic agent returns **free**, so it can load more crates.

- **unload**(?r - robot ?c - crate ?s - supply ?p - person ?k - carrier ?l - location ?startq ?endq - quantity)
  - *duration*: 3 sec (slightly slower w.r.t. the load action due to interaction with person)
  - *condition*
    - *at start*: the robotic agent, the carrier and the person must all be at the same location. The carrier must be carrying the supply crate requested by the person. Of course the robotic agent must be **free**, meaning it is not holding something, in order to unload the crate.
    - *over all*: to prevent more than one crate from being unloaded at the same time, the carrier must have a **startq** number of crates for the entire duration of the action.
  - effect
    - *at start*: the robotic agent will not be **free** anymore as it is holding the crate to deliver to the person who requested it.
    - *at end*: the person has received the desired crate of supply, so he no longer needs it; the number of crates on the carrier has decreased to **endq**, and, lastly, the robotic agent returns **free** since it has delivered the crate.

- **move_to_deliver**(?r - robot ?k - carrier ?src - place ?dst - location) Note: the starting point is a place meaning the agent can start from anywhere in the scenario (depot or location).
  - *duration*: 5 sec
  - *condition*
    - *at start*: the robotic agent and the carrier must be at the same starting position (src).
    - *over all*: the robotic agent holds a crate only when loading or unloading the carrier, in this case it is just delivering a crate so it is **free**.

  - effect
    - *at start*: the robotic agent and the carrier are no more located at src.
    - *at end*: the robotic agent and the carrier arrive at the desired destination (dst).

- **move_to_depot**(?r - robot ?k - carrier ?src - place ?wh - warehouse)
  - *duration*: 5 sec
  - *condition*
    - *at start*: the robotic agent and the carrier are at the same location.
    - *over all*: no other crates will be loaded/unloaded from the carrier during the execution of the action, since only if the carrier is empty (*num_crates(?k - carrier, q0)*) it should return to the depot (wh).
  - effect
    - *at start*: the robotic agent and the carrier are no more at src.
    - *over all*: the robotic agent and the carrier have reached the depot.

4.4 Goal Description

Following the modification introduced in the domain, due to the lack of support by **OPTIC** of *existential-preconditions, negative-preconditions and :disjunctive-preconditions*, also the problem definition has been modified. In particular, since the predicates **needs** and **does_not_need** are used to define what crates the people need, the goal has now been formulated to verify the **does_not_need (person, supply)** condition, meaning the person has received the desired supply crate. So, for example, if the predicate **does_not_need(alice, food)** is true it means that Alice has received the desired crate of food.

5 Problem 4

5.1 Problem Understanding

The last part of the assignment consist in implementing the third problem within the Plansys2 infrastructure [7] [9] [10]. The source code can be found in the **problem4/plansys2_assignment** folder. The **pddl** folder contains a modified version of the pddl code presented as a solution for the previous problem 4. The **launch** folder contains the list of commands to be executed in the *plansys2_terminal* and a modified version of the **launcher.py** file containing the actions defined in the domain file. The **src** folder contains the implementation of each of these fake actions as a **action_node.cpp** files. These nodes are used

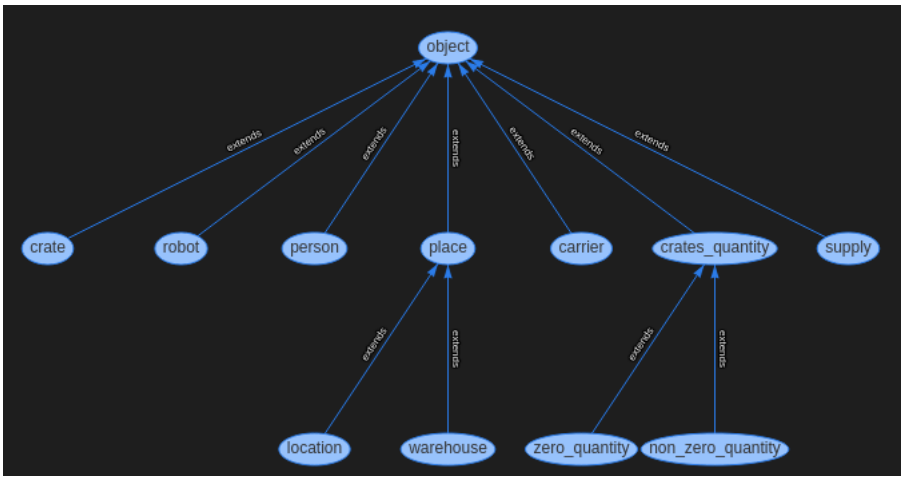

Figure 4: Domain Hierarchy Problem 4

by Plansys2 to simulate the actions execution, thus their duration is the same as the one defined in the domain file:

- **move_to_deliver_action_node** and **move_to_depot_action_node** have a duration of *5 seconds*,
- **unload_action_node** has a duration of *3 seconds*,
- **load_action_node** has a duration of *2 seconds*.

As aforementioned, the **assignment_domain.pddl** file was slightly modified w.r.t. the one defined in the previous problem. In particular, the *constants* symbols were removed from the domain definition, since it seems that *ROS2* cannot correctly interpret them (as reported in [12] and [11]). This has lead to the definition of two new **quantity** types, called **zero_quantity** and **non_zero_quantity**, showed in figure 4. The first is used to model the empty carrier, while the latter is used to model the condition in which the carrier is not empty.

Note that other constants symbols, such as the **depot** and the **robotic agent**, were easily defined via command line using the *set instance* command. These new **quantity** types were introduced to achieve exactly this, define the empty carrier and the max capacity of the carrier via command line.

5.2 Plansys2 Installation

In order to emulate the result reported in this section is necessary to install *Plansys2* on your machine, following the steps below:

- Build **ROS-foxy** from sources as shown in [13]

- Crate a *ps2-ws/src* folder in which all the necessary files will be installed
- Clone the repositories [15] and [8]
- Build the sources:
  - rosdep install -y -r -q –from-paths src –ignore-src –rosdistro foxy
  - colcon build –symlink-install

Once everything has been installed the system should be up and running. Remember to execute *. /opt/ros/foxy/setup.bash* and *. ∼/ps2-ws/install/setup.bash* each time a new terminal is opened in order to properly set up the *ROS2* environment. To facilitate this operation, a script called *ps2-setup.sh* was created (located inside the *problem4/plansys2_assignment* folder).

5.3 Plansys2 Execution

A simple execution of the Plansys2 architecture is shown by the *ps2-assignment-execution* video located in the **problem4/media** folder, which also contains some screenshot of the execution. Notice that, it is possible to access the domain file using the *get domain* command, it is also possible to access the actions, function, predicates and types defined in the domain. The problem predicates and the problem goal are set correctly and are accessible by the architecture as expected, its also possible to obtain a plan using the *get plan* command. But, the execution of the plan does not evolve as expected, in fact, the plan stops at the fist action **load**. This is probably due to the use of program *Oracle VM Virtual Box* for the realization of this assignment. Indeed, due to its virtualization layer that allows to run virtually Ubuntu 20.04 on Windows, it may be the case that the actions required by *plansys2* are not synchronized correctly with the native Windows processor causing the nodes to time out and disrupting the plan execution.

References

[1] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. "Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning". In: *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*. Ed. by Erwin Prassler et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 49–64. ISBN: 978-3-642-25116-0. DOI: 10.1007/978-3-642-25116-0_6. URL: https://doi.org/10.1007/978-3-642-25116-0_6.

[2] Adam Green and other contributors. *ENHSP: Expressive Numeric Heuristic Search Planner*. URL: https://planning.wiki/ref/planners/enhsp. (accessed: 19.07.2022).

[3] Adam Green and other contributors. *OPTIC: Optimising Preferences and Time-Dependent Costs*. URL: https://planning.wiki/ref/planners/optic. (accessed: 19.07.2022).

[4] Adam Green and other contributors. *PDDL 2.1 Domain - Durative Actions*. URL: https://planning.wiki/ref/pddl21/domain#durative-actions. (accessed: 19.07.2022).

[5] Adam Green and other contributors. *PDDL 2.1 Domain - Numeric Fluents*. URL: https://planning.wiki/ref/pddl21/domain#numeric-fluents. (accessed: 19.07.2022).

[6] Adam Green and other contributors. *The Fast Downward Planning System*. URL: https://planning.wiki/ref/planners/fd. (accessed: 19.07.2022).

[7] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[8] Francisco Martìn and colleagues. *PlanSys2/ros2$_p$lanning$_s$ystem$_e$xamples*. URL: https://github.com/PlanSys2/ros2_planning_system_examples.git. (accessed: 19.07.2022).

[9] Francisco Martìn and colleagues. *ROS2 Planning System*. URL: https://plansys2.github.io/index.html. (accessed: 19.07.2022).

[10] Francisco Martın et al. "PlanSys2: A Planning System Framework for ROS2". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - October 1, 2021*. IEEE, 2021.

[11] MostafaGomaa and other contributors. *Pddl domain (:constants) handling and planning*. URL: https://github.com/PlanSys2/ros2_planning_system/pull/139. (accessed: 19.07.2022).

[12] MostafaGomaa and other contributors. *Planning with PDDL constants: constants not parsed*. URL: https://github.com/PlanSys2/ros2_planning_system/issues/135. (accessed: 19.07.2022).

[13] Open Robotics. *ROS 2 Documentation: Foxy*. URL: https://docs.ros.org/en/foxy/Installation.html. (accessed: 19.07.2022).

[14] Marco Roveri. *Assignment for the course Automated Planning Theory and Practice Academic Year 2021-2022*. (accessed: 17.12.2021).

[15] Marco Roveri and other contributors. $fix_compilation_main_on_foxy$. URL: https://github.com/roveri-marco/ros2_planning_system.git. (accessed: 19.07.2022).