

# Automated Planning Theory and Practice Project Report

University of Trento  
MSc in Artificial Intelligence Systems  
Gizem Mert  
Luca Miotto

February 2023

## 1 Introduction

This report details the proposed solution for the assignment on **PDDL** [1] and **PlanSys2** for the “Automated Planning: Theory and Practice” course. The assignment is structured in five sub-problems, each building on the previous one except from problem 5 which directly extends problem 2. In this introductory section, we describe the organization of the delivered archive and the contents of the report.

### 1.1 Archive Organization

The zip archive provided includes five main folders, each corresponding to a problem. Each problem folder containing a solution has three key components:

- domain file
- problem file
- plan file

The domain describes the environment rules and objects. The problem file queries the planner with a specific problem instance, which can be represented in the given domain. The plan file contains the solution returned by planner. The **problem5/plansys2** assignment folder is an exception as it contains the solution for the PlanSys2 part of the assignment and deviates from the standard structure outlined for the other folders.

## 1.2 Report Content

The document is organized according to the following structure, for each problem:

- Description of the problem
- Description of the predicates
- Description of the actions
- Description of the goal.

## 2 Problem 1

### 2.1 Problem Description

The first problem defines an emergency service scenario, that is a logistics problem where a certain amount of people at known locations (and not moving) need to be rescued by the robotic agent. The objective of the planning system is to orchestrate the rescue activities of a set of different robotic agents (restricted to single agent for this assignment) to deliver boxes containing emergency supplies to each person in need. The domain file for this problem includes 10 actions: `fill`, `empty`, `pickup_empty`, `pickup_full`, `move_free`, `move_loaded_empty`, `move_loaded_full`, `deliver_empty`, `deliver_full`, `check_safe`.

Additionally, we define three constants: the `depot`, which is unique (single depot problem) and is present in all subsequent problem instances; the `agent`, which is a robot<sup>1</sup>, and currently happens to have a single member in all problem instances (single agent problem); the `food` `medicine` `tool` content types, which encode the respective types without explicitly creating a type in the domain hierarchy. This is motivated by the need for runtime instance check.

Figure 1 illustrates the domain hierarchy for the defined types. The **Fast Forward** planner was used to generate a solution for this part of the assignment. The resulting plan is illustrated in Figure 2.

### 2.2 Predicates Description

The predicate `at` is used to define the exact location of locatable objects. The predicates `owns` and `instance_of` indicate owner (person or box) that owns some content. The predicate `needs` determines if a person needs for a particular content type (food, tool or medicine). The predicate `carries` is used to indicate if the robot is carrying a given box. The predicates `safe`, `free`, `empty`, `loadable`, `available` respectively check whether a person is safe (*i.e.*, they do not need any content), robot is free (*i.e.*, it does not carry any

---

<sup>1</sup>As requested in the assignment, we use a separate type for robotic agents

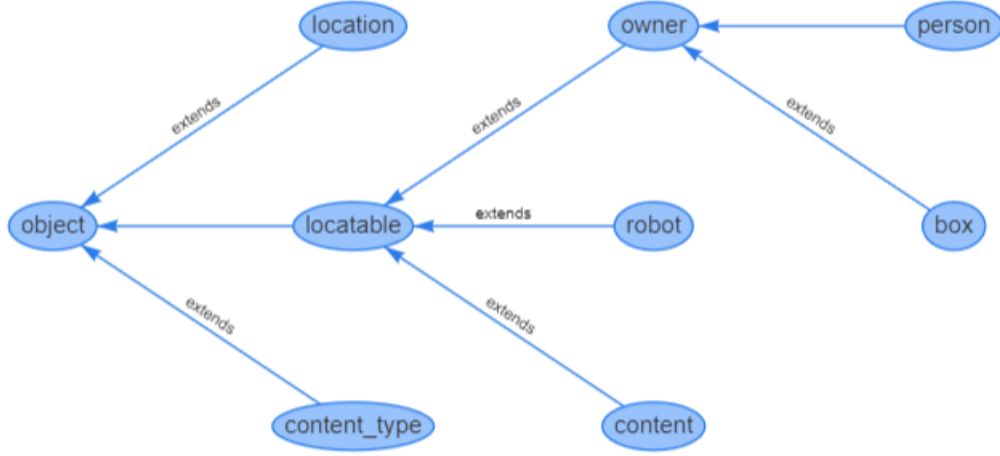


Figure 1: Domain hierarchy for problem 1

boxes), box is empty, box can be loaded with content, content is available for filling some empty box.

## 2.3 Action Description

We propose 10 actions for first problem. The **fill** action allows a robot to fill a single box if the box is empty and content, box and robot are all at the same location. Note that, we assume the robot might fill a box without picking it up. The **empty** action allows a robot to empty a box by dropping the content at the current location, if the box is filled by a content and causing *all* the people at the same location to have the content. Therefore, after emptying, *all* the people at that place who needed that specific content type will be satisfied. We assume the box to be still loaded on the robot after emptying and a robot can empty only the box it carries. The **pickup.empty|full** actions allow a robot to pick up a single box and load it, if it is at the same location as the box, when box is empty or full, respectively. The **move.free** action allows a robot to move to another location if no boxes have been loaded. The **move.loaded.empty|full** actions allow a robot to move to another location if a box has been loaded and it is either empty or has a content, respectively. The **deliver.empty|full** actions allow a robot to deliver a box to a specific person who is at the same location when box is empty or full, respectively. The last action **check\_safe** is used to check person and change status of a person to safe, whenever all needs are satisfied.

## 2.4 Goal Description

We wanted to model people with different needs, so the goal was defined as a combination of people needing either only food/medicine/tool, or a subset of them, or being already

safe (*i.e.*, people who don't need anything). In order to determine if needs of all people have been met, we check the `safe` predicate. We define *2 boxes*, *3 locations*, *6 contents* and *5 people* for our scenario. Initially, all contents, robot, boxes and one safe person are at depot. Boxes are empty. It is worth mentioning that two of the people in need are at same location out of depot. We achieved our goal by using **Fast Forward** planner. The result is shown in Figure 2.

```

0.00000: (PICKUP_EMPTY AGENT B2 DEPOT)
0.00100: (FILL AGENT B2 C5 MEDICINE DEPOT)
0.00200: (MOVE_LOADED_FULL AGENT B2 C5 DEPOT L1)
0.00300: (EMPTY AGENT B2 C5 MEDICINE L1)
0.00400: (CHECK_SAFE GIZEM)
0.00500: (MOVE_LOADED_EMPTY AGENT B2 L1 DEPOT)
0.00600: (FILL AGENT B1 C4 MEDICINE DEPOT)
0.00700: (FILL AGENT B2 C2 FOOD DEPOT)
0.00800: (MOVE_LOADED_FULL AGENT B2 C2 DEPOT L3)
0.00900: (EMPTY AGENT B2 C2 FOOD L3)
0.01000: (MOVE_LOADED_EMPTY AGENT B2 L3 DEPOT)
0.01100: (FILL AGENT B2 C6 TOOL DEPOT)
0.01200: (MOVE_LOADED_FULL AGENT B2 C6 DEPOT L3)
0.01300: (EMPTY AGENT B2 C6 TOOL L3)
0.01400: (CHECK_SAFE INDIRA)
0.01500: (MOVE_LOADED_EMPTY AGENT B2 L3 DEPOT)
0.01600: (FILL AGENT B2 C1 FOOD DEPOT)
0.01700: (MOVE_LOADED_FULL AGENT B2 C1 DEPOT L2)
0.01800: (EMPTY AGENT B2 C1 FOOD L2)
0.01900: (MOVE_LOADED_EMPTY AGENT B2 L2 DEPOT)
0.02000: (FILL AGENT B2 C3 MEDICINE DEPOT)
0.02100: (MOVE_LOADED_FULL AGENT B2 C3 DEPOT L3)
0.02200: (EMPTY AGENT B2 C3 MEDICINE L3)
0.02300: (CHECK_SAFE SAVANNAH)
0.02400: (DELIVER_EMPTY AGENT B2 SAVANNAH L3)
0.02500: (MOVE_FREE AGENT L3 DEPOT)
0.02600: (PICKUP_FULL AGENT B1 C4 MEDICINE DEPOT)
0.02700: (MOVE_LOADED_FULL AGENT B1 C4 DEPOT L2)
0.02800: (EMPTY AGENT B1 C4 MEDICINE L2)
0.02900: (CHECK_SAFE DARIA)
0.03000: (MOVE_LOADED_EMPTY AGENT B1 L2 DEPOT)

```

Figure 2: Plan for problem 1

## 3 Problem 2

### 3.1 Problem Description

The second problem is a more complex version of the first one. It involves a new locatable type called a **carrier**, which acts as a container for the agent and can hold multiple boxes simultaneously. This introduces the additional challenge of managing the inventory of the carrier, separate from the previous problem. Additionally, the robotic agent and carrier are attached, forming a single unit. This means that they move together and the robotic agent is responsible for loading/unloading boxes and moving the carrier between locations. Figure 3 illustrates the domain hierarchy for the defined types. The **ENHSP-2020** (Expressive Numeric Heuristic Search Planner) planner was used to generate a solution for this part of the assignment.

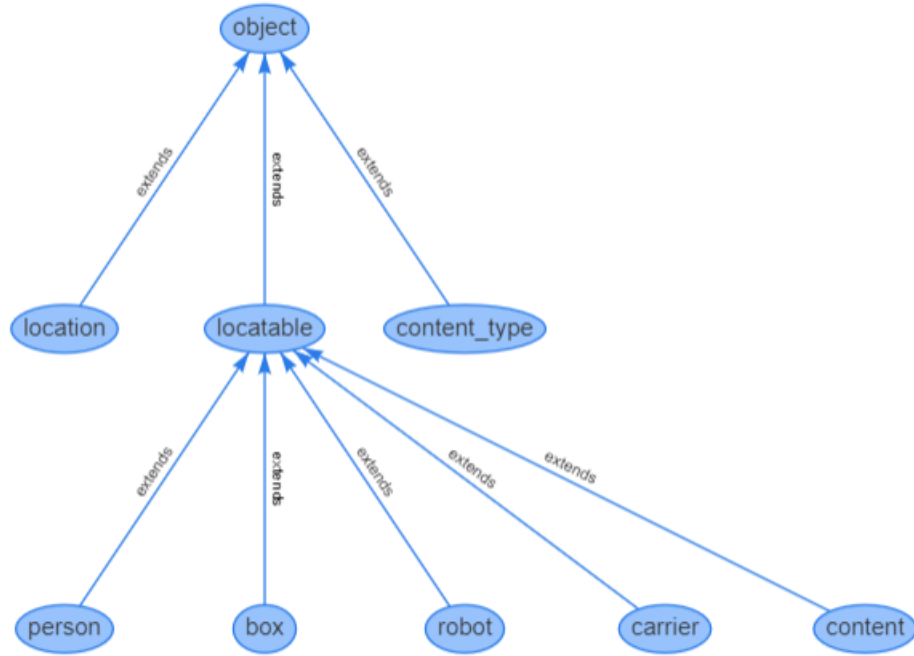


Figure 3: Domain hierarchy for problem 2

### 3.2 Predicates Description

The predicate **at** is used to define the exact location of locatable object. The predicates **contains** and **instance\_of** indicate box contains the content and content type. The predicate **needs** determines if a person needs for a particular content type (food, tool or medicine). The predicate **pulls** determines robot is pulling the carrier. The predicate **carries** is used to indicate the carrier is carrying a particular box. The predicate

`is_emptying_at` is used to indicate that robot is emptying box at given location. The predicates `busy`, `empty`, `loadable`, `available` respectively check whether the robot is busy (*i.e.*, it is performing non-atomic action), box is empty, box can be loaded with content, content is available for filling some given box.

We added the following functions to track the carrier during actions:

- `max_load`: max amount of boxes allowed for a given carrier (*fixed* and *problem-dependent*, since it can be set in the problem file)
- `count_load`: counts the boxes loaded on the carrier
- `count_needs`: counts content types needed by a given person
- `count_needed`: counts people who need a given content type
- `count_needed_at`: counts people who need a given content type at a specific location

It is worth specifying that `count_load`, `count_needs`, `count_needed`, `count_needed_at` are initialized for each item in the problem file, and are expected to be *consistent* with each other and with respect to the chosen `max_load` upper bound.

### 3.3 Action Description

We propose 6 actions in second problem. The `fill` action allows a robot to fill a single box if the box is empty and content, box and agent are all at the same location. Once again, we assume the robot might fill a box even if it is not loaded. This design would allow for a robot filling an empty box loaded by another robot, in a multi-agent scenario. The `empty` action allows a robot to empty a box by dropping the content at the current location if the box is filled by a content and causing *one* person at that place to have their need satisfied. If multiple persons at the same location need the content, the robot can keep emptying (*i.e.*, repeating the `empty` action at the same location with different target people) until all people at that location have been satisfied (*i.e.*, until `check_empty` action is invoked). The amount of content in a box is assumed to be big enough for many people. Therefore, this action is *not* intended as an atomic action. Moreover, we once again assume the box to be still loaded on the robot's carrier after emptying and robot can empty only a box carried by its own carrier. The `check_empty` action allows to change status of a box to empty (*i.e.*, all people at location needing that content type were successfully served). Also, after emptying, the position of content actively changes, since it's not in the box anymore. The `pickup` action allows a robot to pick up a single box and load it on the carrier, if it is at the same location as the box. The `move` action allows a robot to move to another location regardless from robot has anything attached or not. The `deliver` action allows a robot to deliver a box to a specific person who is at the same location, and after deliver, the effect actively changes position of box, since it's not on the carrier anymore.

For action definitions, there is *no need* to check for position of other locatables since we check they're all attached to each other.

### 3.4 Goal Description

We wanted to model people with different needs, so the goal was defined as a combination of people needing either only food/medicine/tool, or a subset of them, or being already safe (*i.e.*, people who don't need anything). To determine that the needs of all people are satisfied, we check if the `count_needs` predicate is equal to zero. Also we add goal to make agent return to depot to be ready for new tasks.

We define *1 carrier, 5 boxes, 3 location, 6 content, 5 people* for our scenario. Initially, all contents, robot, boxes and one safe person are at depot. Boxes are empty. It is worth mentioning that two of the people in need are at same location out of depot. We defined the maximum loadable amount to carrier as 4. We defined also counts of needs of people at particular locations. We achieved our goal by using **ENHSP-2020** planner. The result is shown in Figure 4.

## 4 Problem 3

### 4.1 Problem Description

The problem at hand requires us to apply **Hierarchical Task Networks** (HTN) to the scenario described in problem 2. To tackle this issue, we must incorporate the use of `:tasks` and `:methods`. In an HTN, tasks are arranged in a hierarchical structure where the higher-level tasks represent the overall goal of the system, and the lower-level tasks represent the individual actions that need to be taken to achieve the goal. The system uses methods to break down higher-level tasks into lower-level ones until the tasks are simple enough to be executed. Figure 5 illustrates the domain hierarchy for the defined types.

### 4.2 Predicates Description

Most of the predicates remain unchanged with respect to what already discussed for problem 2. The added constant for this problem is **zero**, which is used to represent the null quantity of contents. In addition to the ones inherited from the previous problem, we add the following:

- **safe**: checks if all person's needs are satisfied
- **not\_busy**: checks if robot is busy performing some non-atomic action
- **succ**: assigns a quantity as successor of another quantity
- **leq**: assigns a quantity as less than or equal to another quantity

```

0.00000: (fill agent b1 c5 medicine depot)
1.00000: (fill agent b3 c1 food depot)
2.00000: (fill agent b4 c6 tool depot)
3.00000: (pickup agent cr1 b2 depot)
4.00000: (pickup agent cr1 b1 depot)
5.00000: (pickup agent cr1 b3 depot)
6.00000: (pickup agent cr1 b4 depot)
7.00000: (fill agent b2 c3 medicine depot)
8.00000: (move agent depot l2)
9.00000: (empty agent cr1 b1 c5 medicine daria l2)
10.00000: (check_empty agent cr1 b1 c5 medicine l2)
11.00000: (move agent l2 depot)
12.00000: (fill agent b1 c4 medicine depot)
13.00000: (move agent depot l1)
14.00000: (empty agent cr1 b2 c3 medicine gizem l1)
15.00000: (check_empty agent cr1 b2 c3 medicine l1)
16.00000: (move agent l1 depot)
17.00000: (fill agent b2 c2 food depot)
18.00000: (move agent depot l2)
19.00000: (empty agent cr1 b2 c2 food daria l2)
20.00000: (check_empty agent cr1 b2 c2 food l2)
21.00000: (move agent l2 l3)
22.00000: (empty agent cr1 b3 c1 food indira l3)
23.00000: (empty agent cr1 b3 c1 food savannah l3)
24.00000: (check_empty agent cr1 b3 c1 food l3)
25.00000: (empty agent cr1 b4 c6 tool indira l3)
26.00000: (empty agent cr1 b4 c6 tool savannah l3)
27.00000: (check_empty agent cr1 b4 c6 tool l3)
28.00000: (empty agent cr1 b1 c4 medicine savannah l3)
29.00000: (check_empty agent cr1 b1 c4 medicine l3)
30.00000: (move agent l3 depot)

```

Figure 4: Plan for problem 2

Moreover, thanks to the use of a `quantity` type, we managed to apply numerical expressions without actually implementing numerical values. This choice is motivated by the restrictions of most hierarchical planners, not allowing for the use of numerics.

### 4.3 Action Description

For this task, the same set of actions - `fill`, `check_empty`, `pickup`, `move` and `deliver` - used in the second problem are exploited once again. However, three new actions have been added to the list:



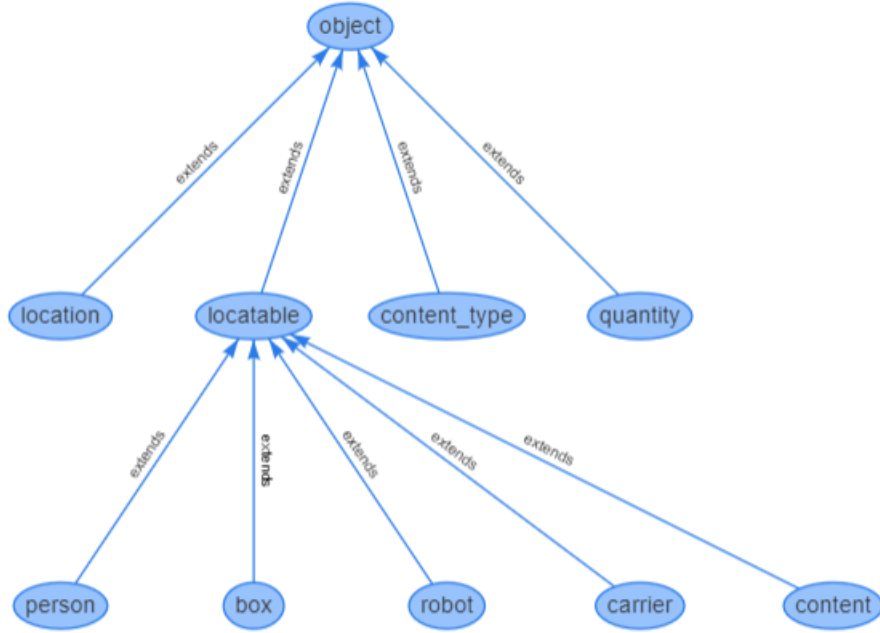


Figure 5: Domain hierarchy for problem 3

- **start\_empty**: This action enables the robot to empty a box by releasing its contents at the current location, causing the people at the same location to obtain the content.
- **continue\_empty**: This action serves the same purpose as the **start\_empty** action, but differs in that the robot must be busy with emptying the box before taking the action, whereas emptying the box is an effect of the **start\_empty** action.
- **check\_safe**: changes status of a person if all needs are satisfied.

#### 4.3.1 Tasks

We defined tasks **t\_rescue** and **t\_return** as higher-level tasks, which are used in different methods. The first one represents the task of rescuing a person, which means bringing them a content they needed. Note that, if multiple contents are needed by a person, then the problem will contain multiple sub-tasks of the form `(task_i (t_rescue agent person))`. Thanks to the preconditions enforced in the domain, the planner will automatically “understand” which of the person’s needs are yet to be satisfied, and thus the person will be rescued accordingly. Of course, it is required that the sub-tasks are defined consistently with the other definitions. Finally, the second task is the usual additional goal we enforced in all problems, forcing the robot to return to depot after completing all the main tasks.

### 4.3.2 Methods

We created 17 different methods which try to achieve high-level tasks by specifying low-level actions in a given order. The implemented methods are the following:

- `m_rescue_1`: perform a rescue operation by emptying a box for one person
- `m_rescue_2`: perform a rescue operation for by emptying a box for two persons
- `m_go_rescue_1`: perform a rescue operation by moving and emptying for one person
- `m_go_rescue_2`: perform a rescue operation by moving and emptying for two persons
- `m_load_1_and_go_rescue_1`: perform a rescue operation by picking, filling one box, moving and emptying for one person
- `m_load_2_and_go_rescue_1`: perform a rescue operation by picking, filling two boxes, moving and emptying for one person
- `m_load_1_and_go_rescue_2`: perform a rescue operation by picking, filling one box, moving and emptying for two persons
- `m_load_2_and_go_rescue_2`: perform a rescue operation by picking, filling two boxes, moving and emptying for two persons
- `m_go_load_1_and_go_rescue_1`: perform a rescue operation by moving back to depot, picking, filling one box, moving and emptying for one person
- `m_go_load_1_and_go_rescue_2`: perform a rescue operation by moving back to depot, picking, filling one box, moving and emptying for two people
- `m_fill_1_and_go_rescue_1`: perform a rescue operation by filling one box, moving and emptying for one person
- `m_fill_2_and_go_rescue_1`: perform a rescue operation by filling two boxes, moving and emptying for one person
- `m_fill_1_and_go_rescue_2`: perform a rescue operation by filling one box, moving and emptying for two people
- `m_fill_2_and_go_rescue_2`: perform a rescue operation by filling two boxes, moving and emptying for two persons
- `m_go_fill_1_and_go_rescue_1`: perform a rescue operation by moving, filling one box, moving and emptying for one person
- `m_go_fill_1_and_go_rescue_2`: perform a rescue operation by moving, filling one box, moving and emptying for two persons

- **m\_return\_to\_depot**: perform a return operation by moving back to depot after finishing whole rescue operations

It is worth noticing that we only defined a *restriction* of all the possible methods which could be implemented in such scenario. In particular, note that we did not use **deliver** actions in these methods. Moreover, we limited the number of boxes loaded/filled at each sequence of actions to a maximum of two. The global maximum number of loaded boxes remains fixed and problem dependent (*e.g.*, four, in this particular problem instance) but they cannot be loaded/filled while executing the same method. However, they can be loaded/filled in different methods while solving the same problem. Nevertheless, by iterating with the same construction logic, more methods allowing for any number of loading/filling operations (up to the maximum allowed, according to the problem) could be easily defined. Similarly, the number of persons rescued at the same place has been limited to a maximum of two. Once again, such limit could be easily overcome.

## 4.4 Goal Description

We define *1 carrier*, *2 boxes*, *2 location*, *6 content*, *5 quantities* (including **zero**) and *2 people* for our scenario. Initially, all contents, robot, carrier, boxes and one safe person are at depot and boxes are empty, while 2 people are out of depot. We defined also counts of needs of people at particular locations. We defined the maximum loadable amount to carrier as 4.

In defining the problem, the higher-level task of **t\_rescue** is applied as frequently as the combined number of requirements for all people.

In this particular case, the computational complexity of the problem is a limiting factor, as it would take an excessive amount of time to find a solution by exploring the full state space. Therefore, a restriction of the problem involving fewer people and boxes was used for the demonstration. We achieved our goal by using **PANDA** (Planning and Acting in a Network Decomposition Architecture) planner. The result is shown in Figure 6.

# 5 Problem 4

## 5.1 Problem Description

This task is the extension of problem 2, with the introduction of **:durative-actions**. Figure 7 illustrates the domain hierarchy for the defined types. **POPF** (Partial Order Planning Forwards) planner was used to generate a solution for this part of the assignment.

## 5.2 Predicates Description

Most of the predicates remain unchanged with respect to the ones already discussed for problem 2. The added predicates for this problem are **safe**, used to check whether needs

```

0: (pickup agent cr1 b1 depot zero q1 q4)
1: (pickup agent cr1 b2 depot q1 q2 q4)
2: (fill agent b1 c5 medicine depot)
3: (fill agent b2 c3 medicine depot)
4: (move agent depot l1)
5: (start_empty agent cr1 b1 c5 medicine gizem l1 q1 zero q1 zero)
6: (check_empty agent cr1 b1 c5 medicine l1 zero)
8: (move agent l1 depot)
9: (fill agent b1 c1 food depot)
10: (move agent depot l2)
11: (start_empty agent cr1 b2 c3 medicine daria l2 q2 q1 q1 zero)
12: (check_empty agent cr1 b2 c3 medicine l2 zero)
13: (start_empty agent cr1 b1 c1 food daria l2 q1 zero q1 zero)
14: (check_empty agent cr1 b1 c1 food l2 zero)
15: (move agent l2 depot)

```

---

Figure 6: Plan for problem 3

all needs of a person are satisfied, and `not_busy`, used to enforce action sequentiality and avoid dangerous overlapping.

### 5.3 Action Description

The actions `fill`, `check_empty`, `pickup`, `move` and `deliver` are used again for this task same as second problem. We added three new actions as listed:

- `start_empty`: allows robot to empty a box by releasing its contents at the current location, and causing *one* person at the same location to have the content.
- `continue_empty`: has the same purpose as the previous one, but differs from it by initial situation of the robot as condition. This action allows the robot to continue the emptying action at the same location, but serving *another* person who has the same need as the one previously served.
- `check_safe`: changes status of a person if all needs are satisfied.

We chose a constant duration of 3 for the actions `fill`, `start_empty` and `continue_empty`; a duration of 1 for the actions `check_empty` and `check_safe`; a duration of 2 for actions `deliver` and `pickup`; a duration of 5 for the action `move`.

For action definitions, there is *no need* to check for position of other locatables since we check they're all attached to each other.

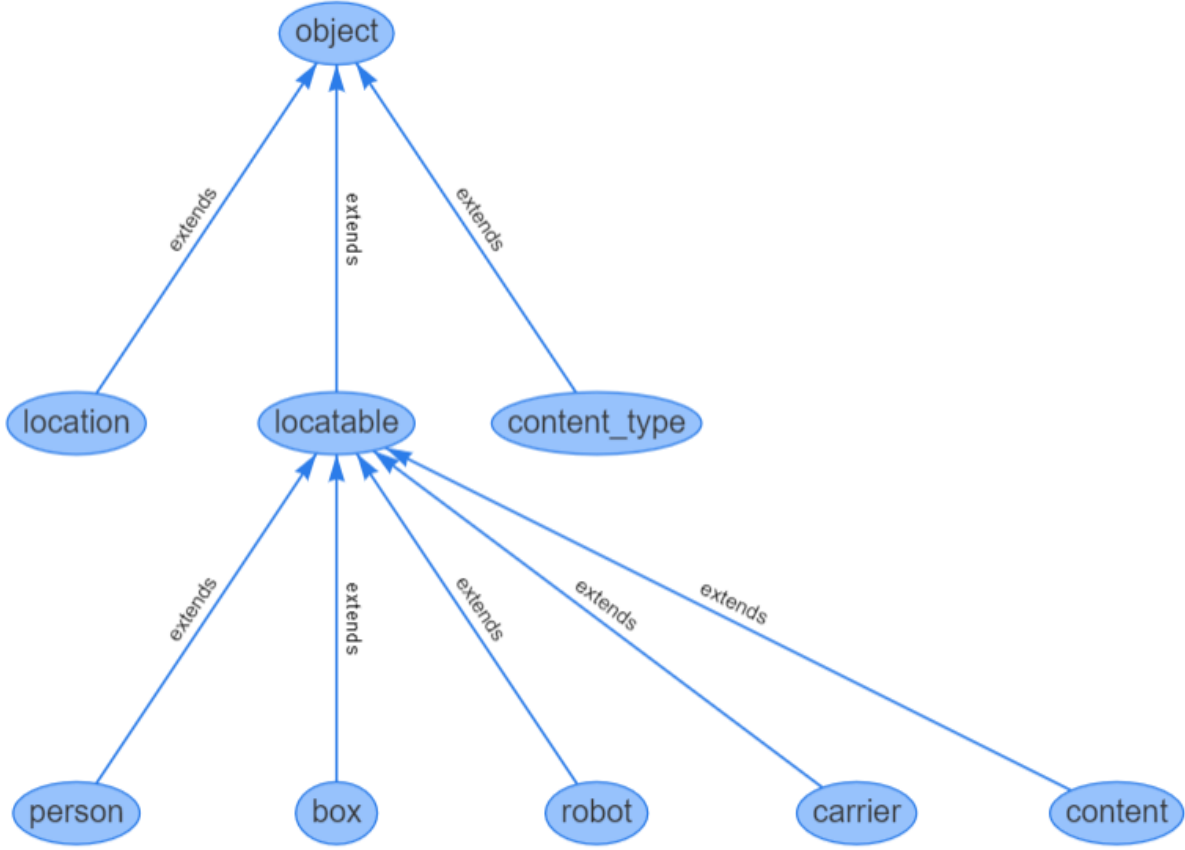


Figure 7: Domain hierarchy for problem 4

## 5.4 Goal Description

As the planner does *not* support expressing goals with disjunctive or negative preconditions, we adopted a different approach where the goal is defined as a state in which all individuals are **safe**, which means they have been served by a robotic agent for all their needs. Thus, the successful service of every person is now the representation of the goal.

Once again, the computational complexity of the problem, together with our scarce hardware resources, has forced us to query the planner with a restricted version of our main problem. We define *1 carrier*, *1 boxes*, *2 locations*, *3 contents* and *2 people* for our scenario. Initially, all contents, robot, carrier and boxes are at depot, and boxes are empty. Both of the people involved are out of depot and need for rescue. We defined also counts of needs of people at particular locations. We achieved our goal by using **POPF** (Partial Order Planning Forwards) planner. The resulting plan, with durations, is shown in Figure 8.

```

0.000: (fill agent b1 c3 medicine depot) [3.000]
3.001: (pickup agent cr1 b1 depot) [2.000]
5.002: (move agent depot l1) [5.000]
10.003: (start_empty agent cr1 b1 c3 medicine gizem l1) [3.000]
13.004: (check_empty agent cr1 b1 c3 medicine l1) [1.000]
14.005: (check_safe agent gizem) [1.000]
15.006: (move agent l1 depot) [5.000]
20.007: (fill agent b1 c1 food depot) [3.000]
23.008: (move agent depot l2) [5.000]
28.009: (start_empty agent cr1 b1 c1 food daria l2) [3.000]
31.010: (check_empty agent cr1 b1 c1 food l2) [1.000]
32.011: (move agent l2 depot) [5.000]
37.012: (fill agent b1 c4 medicine depot) [3.000]
40.013: (move agent depot l2) [5.000]
45.014: (start_empty agent cr1 b1 c4 medicine daria l2) [3.000]
48.015: (check_empty agent cr1 b1 c4 medicine l2) [1.000]
49.016: (check_safe agent daria) [1.000]
50.017: (move agent l2 depot) [5.000]

```

Figure 8: Plan for problem 4

## 6 Problem 5

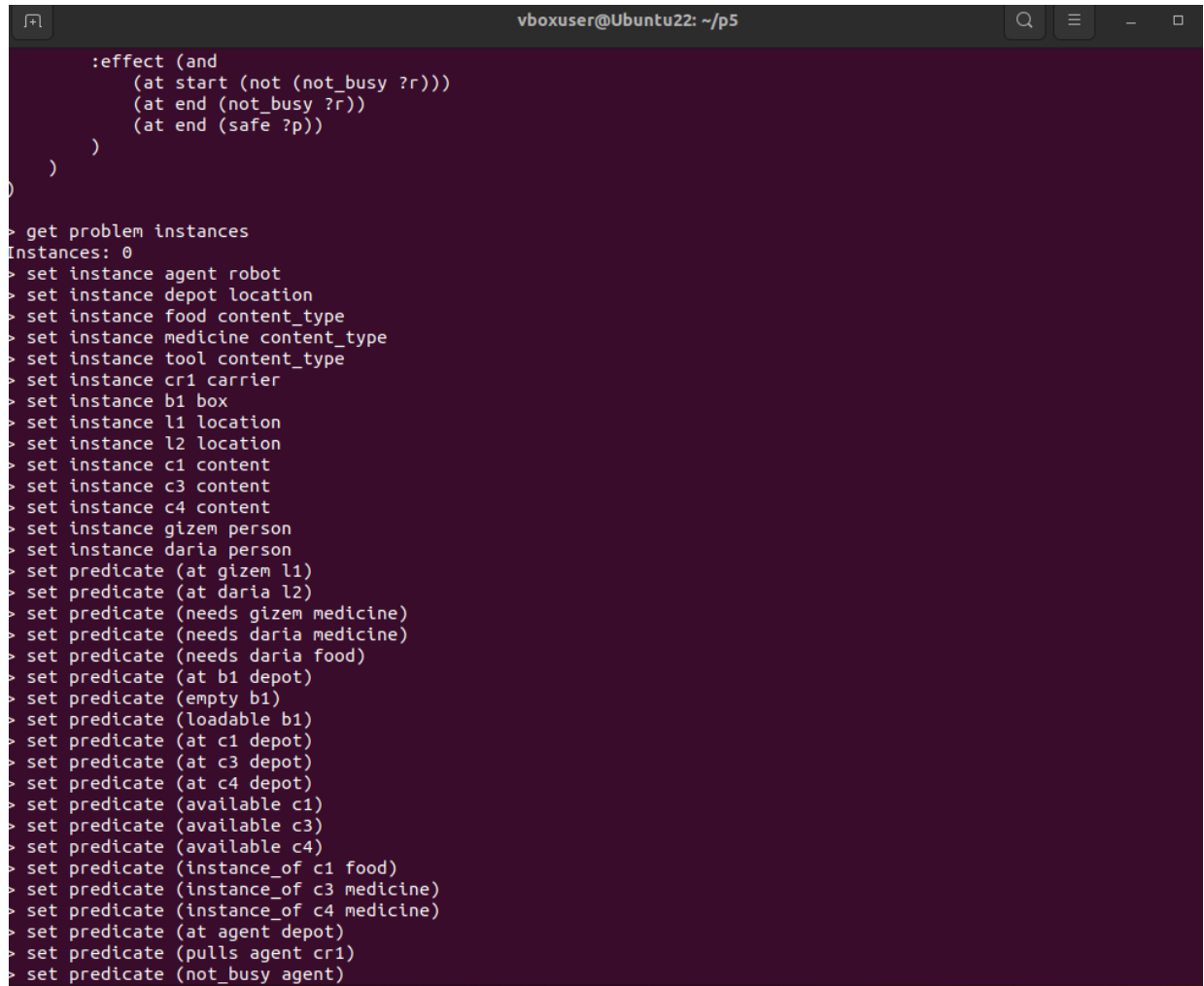
The final step of the assignment involves integrating the fourth problem into the **PlanSys2** infrastructure [2] [3]. The source code for this part can be found in the `problem5/plansys2.assignment` folder. In this folder, the `pddl` directory contains domain code from problem 4, while the `launch` directory includes a modified version of the `launcher.py` file and a list of commands to execute in the PlanSys2 terminal. Finally, the `src` folder contains the implementation of each simulated action as an `action node.cpp` file.

We defined the problem via command line using the `set instance`, `set predicate` and `set function` commands. The plan was executed from `problem.plan` file which was found by the **POPF** planner.

### 6.1 Installation and Execution in PlanSys2

To install PlanSys2 on your machine, you need to build ROS, create a specific folder for the necessary files, clone repositories, build sources, and execute certain commands. Once everything is installed, execute `. /opt/ros/foxy/setup.bash` and `. ~/ps2-ws/install/setup.bash` each time a new terminal is opened to properly set up the ROS2 environment. The `problem5/plansys2.assignment/media` folder contains several screenshots that demonstrate a basic execution in the PlanSys2 architecture. By using the `get domain` command, it is possible to access the domain file, actions, functions, predicates, and types

defined in the domain. The problem predicates and goal are correctly set and accessible, and the `get plan` command can be used to obtain a plan.

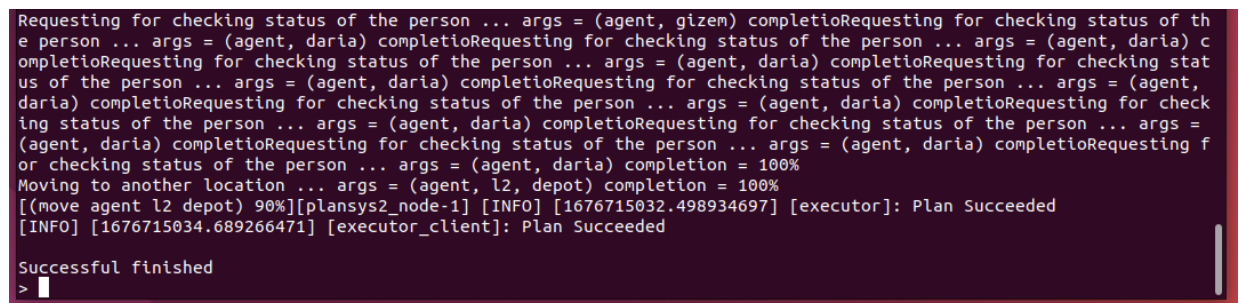


```

vboxuser@Ubuntu22: ~/p5
:effect (and
  (at start (not (not_busy ?r)))
  (at end (not_busy ?r))
  (at end (safe ?p))
)
)
)
> get problem instances
Instances: 0
> set instance agent robot
> set instance depot location
> set instance food content_type
> set instance medicine content_type
> set instance tool content_type
> set instance cr1 carrier
> set instance b1 box
> set instance l1 location
> set instance l2 location
> set instance c1 content
> set instance c3 content
> set instance c4 content
> set instance gizen person
> set instance daria person
> set predicate (at gizen l1)
> set predicate (at daria l2)
> set predicate (needs gizen medicine)
> set predicate (needs daria medicine)
> set predicate (needs daria food)
> set predicate (at b1 depot)
> set predicate (empty b1)
> set predicate (loadable b1)
> set predicate (at c1 depot)
> set predicate (at c3 depot)
> set predicate (at c4 depot)
> set predicate (available c1)
> set predicate (available c3)
> set predicate (available c4)
> set predicate (instance_of c1 food)
> set predicate (instance_of c3 medicine)
> set predicate (instance_of c4 medicine)
> set predicate (at agent depot)
> set predicate (pulls agent cr1)
> set predicate (not_busy agent)

```

Figure 9: Problem setting



```

Requesting for checking status of the person ... args = (agent, gizen) completioRequesting for checking status of th
e person ... args = (agent, daria) completioRequesting for checking status of the person ... args = (agent, daria) c
ompletioRequesting for checking status of the person ... args = (agent, daria) completioRequesting for checking stat
us of the person ... args = (agent, daria) completioRequesting for checking status of the person ... args = (agent,
daria) completioRequesting for checking status of the person ... args = (agent, daria) completioRequesting for check
ing status of the person ... args = (agent, daria) completioRequesting for checking status of the person ... args =
(agent, daria) completioRequesting for checking status of the person ... args = (agent, daria) completioRequesting f
or checking status of the person ... args = (agent, daria) completion = 100%
Moving to another location ... args = (agent, l2, depot) completion = 100%
[(move agent l2 depot) 90%][plansys2_node-1] [INFO] [1676715032.498934697] [executor]: Plan Succeeded
[INFO] [1676715034.689266471] [executor_client]: Plan Succeeded

Successful finished
>

```

Figure 10: Plan Succeeded

```

> run plan-file problem.plan
The plan read from "problem.plan" is
0:      (fill agent b1 c3 medicine depot)      [3]
3.001:  (pickup agent cr1 b1 depot)            [2]
5.002:  (move agent depot l1)                  [5]
10.003: (start_empty agent cr1 b1 c3 medicine gizem l1) [3]
13.004: (check_empty agent cr1 b1 c3 medicine l1) [1]
14.005: (check_safe agent gizem)              [1]
15.006: (move agent l1 depot)                  [5]
20.007: (fill agent b1 c1 food depot)          [3]
23.008: (move agent depot l2)                  [5]
28.009: (start_empty agent cr1 b1 c1 food daria l2) [3]
31.01:  (check_empty agent cr1 b1 c1 food l2)   [1]
32.011: (move agent l2 depot)                  [5]
37.012: (fill agent b1 c4 medicine depot)      [3]
40.013: (move agent depot l2)                  [5]
45.014: (start_empty agent cr1 b1 c4 medicine daria l2) [3]
48.015: (check_empty agent cr1 b1 c4 medicine l2) [1]
49.016: (check_safe agent daria)               [1]
50.017: (move agent l2 depot)                  [5]
[plansys2_node-1] [INFO] [1676648540.817142491] [executor]: Action fill timeout percentage 0000
[plansys2_node-1] [INFO] [1676648540.904677115] [executor]: Action pickup timeout percentage 000000
[plansys2_node-1] [INFO] [1676648540.929496435] [executor]: Action move timeout percentage 0000
[plansys2_node-1] [INFO] [1676648541.050727653] [executor]: Action start_empty timeout percentage -1.000000
[plansys2_node-1] [INFO] [1676648541.149033947] [executor]: Action check_empty timeout percentage -1.000000
[plansys2_node-1] [INFO] [1676648541.224803879] [executor]: Action check_safe timeout percentage -1.000000
[plansys2_node-1] [INFO] [1676648541.240558401] [executor]: Action move timeout percentage

```

Figure 11: Plan for problem 5

## References

- [1] Cristian Muise and other contributors. PLANUTILS, 2021 General library for setting up linux-based environments for developing, running, and evaluating planners. <https://github.com/AI-Planning/planutils>.
- [2] Francisco Martin and colleagues. PlanSys2 Tutorials, 2021. <https://intelligentroboticslab.gsync.urjc.es/ros2planningsystem.github.io/tutorials/index.html>
- [3] Francisco Martin, Jonatan Gines, Francisco J. Rodriguez, and Vicente Matellan. PlanSys2: A Planning System Framework for ROS2. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS 2021.