Automated Planning Report

Davide De Martini, Steyn Mulder

I. Introduction

This report aims to explain the design choices and outcomes of the automated planning final project. It is organized as follows: first, a general problem description is provided, highlighting the key assumptions made during the project's development. The following five sections focus on the specific problem specifications and their explanations. Subsequently, the results are presented, followed by a discussion and conclusion. The source code can be found at the link https://github.com/davidedema/automated_planning.

II. Problem Description

This section presents the scenario along with the general assumptions made across all five problems. As the scenario definition is already thoroughly detailed in the assignment track, it will only be briefly addressed here.

We have to model a healthcare domain in which agents have to deliver supplies (e.g. bandages, syringes) to medical units (e.g. radiology, emergency) and to accompany patients to medical units. The hospital is composed of different locations and agents can navigate through adjecent locations.

In order to model this scenario, we have adopted the *do it simple* philosophy, PDDL could be tweaked with a lot of features, the problem is that not all planners support all of them. So in order to be compatible with the majority of the planners, the requirements that we have used are `:typing`, `:strips` and `:durative-actions` (for temporal planning problem).

A more in depth discussion on the different modelling choices is done in the next five sections.

III. Problem 1

In the first part of the project, our objective was twofold. Firstly, we wanted to model a robot system that could load boxes with supplies within a centralized warehouse location and transport those boxes to other locations, and more specifically to certain units within these locations. Secondly, we wanted to model a robot system that can bring patients from a centralized entrance to their desired unit.

A. General

For the overall model of the hospital that was designed, the choice was made to model locations as a container for units, thus a location can contain several units, but a unit can only be part of one location.

The adjecency of locations signifies between which locations an agent can move. This is modeled with the predicate `adjacent`. And the containment of a unit within a location is modeled with the predicate `belongs`. Next, we modeled the localization of the robot such that it can only be in one location at any given moment. This is modeled with the predicate `atl`.

B. Supply delivery

For the supply delivery, first we created a child of the robot object called box_robot. This subtype will have the privileges to fulfill supply delivery tasks. We then modeled how the robot is holding a box. We did this using the predicate `has`. To signal that a robot is not holding a box, we also made the predicate `free`.

Then we modeled the box, which we assume can contain only one supply. With the predicate `at` we say in which location the box is, with `has_supply` we say the box contains a certain supply object, and with `empty` we now signal that the box has no supplies inside it.

For the supplies we modeled them as individual objects. We also made the assumption that the warehouse location has both enough supplies and boxes to service all incoming requests and the warehouse is the only location at which supplies can be loaded into boxes. We modeled this with the predicate `has_supply_at`. Conversely, to have units make requests for supplies, we use `has_unit`, which means a possible goal could be `(has_unit bandage unit1)`.

The general flow for fulfilling an order for supply delivery would then look something like this. The box_robot starts moving towards the warehouse location using the action `move` to move between adjacent locations, possibly executing this action multiple times, until it is at the warehouse. Then it uses the action `load_box`, which is a combined action resulting in a box being filled with a supply and the robot now carrying said box. After this, the robot will again move between locations until it is at the location containing the correct unit for delivery. This now happens with the action `move_box` instead. Once it is at the correct location, it uses action `empty_box`, which again is a combination of the robot unloading the box from itself, taking the supply out of the box, and handing the supplies over to the correct unit.

C. People Accompaniment

As with the `box_robot`, for this task we created a new subtype of robot for the job, called a `guide_robot`. We have made the assumption that with this type of work, one robot is able to assist multiple people at the same time, regardless of whether they need to go to the same unit.

For the person object, we modeled its intent with the predicate `in`, so a goal could look like `(in person1 unit1)`. To specify in which location the person is, we used `inl`. Lastly, to say that a guide_robot is helping a person we use the predicate `accompanying`.

The flow for guiding someone looks like this. The guide_robot and the person should first be in the same location, which we always assume to be the entrance. If the robot is not there, it first uses the `move` action, just as was described in the previous subsection. Then a handshake is done between the robot and the person, which we model with the action `start_accompany`. This means that the person says to which unit it needs to go, the robot starts `accompanying` the person, and the person follows by always taking on the same location as the robot that is helping it. Again, the `move` action is used to move to the location that contains the right unit. At this point, the procedure is finished with the combined action `accompany`, which takes the person to their desired unit and stops the robot from accompanying the person.

D. Additional Exploration: Supplies as Predicates

When we first started developing the domain, we implemented each instance of a supply as a seperate predicate/action, in the form of: `has_bandage`, `has_bandage_at`, `has_unit_bandage`, and the actions `load_box_bandage`, `empty_box_bandage`, and then on to `has_scalpel`, et cetera. We later switched to the implementation we have now, where supplies are modeled as objects. We did this for several reasons. Firstly, it was an intensive task to make all these predicates and actions for multiple supplies. Second, it would be difficult to translate this in the later stages of the project, when we need to work with an HTN and temporal planning. Lastly, by specifying the supplies as predicates, we make them domain dependent, instead of problem specific. In a real-world scenario, where this system could be used by several hospitals, all with their own list of supplies, this approach is much less feasible and scalable. Nevertheless, we were curious about the performance differences between the two, which is why we made a comparison in the Results section VIII-A.

IV. Problem 2

For this part of the assignment, our objective was to give each of the box_robots a problem-specific capacity, i.e. a certain number of boxes it can carry at once, as well as implement additional methods for robots to move around.

A. Robot Carrier

To implement robot capacity, we opted to give each robot a carrier object. This carrier in turn has a number of slots, specified in the problem file. We model the slots such that exactly one box is placed in exactly one slot. In this way we can model different capacity for different robots. An additional predicate for the box_robot is `has`, which states that the robot has a specific carrier. The carrier location is stated by `at_location_carrier`. We keep track of which slots belong to which carriers using `has_slot`. The boxes now no longer belong to the robot, but instead to a specific slot on a carrier, signified by `contains`. To tell whether or not a box can be placed on a slot, we use the predicate `empty_slot`.

The flow of fulfilling an order now looks like this. Movement is now always done with a carrier, so the action for moving to the warehouse is `move_carrier`. Then the `load_box` action is now replaced by the `load_carrier` action. It still works the same as before, with the addition of the slot on the carrier now being filled by the box. The unloading is also similar, being done by the `unload_carrier` action. Also now, the additional effects are that the slot in which the box was carried over is now empty again.

B. Robot Types

We have made further subtypes of box_robots, to model different kinds of movement behavior and accessibility. Firstly, we have the flying_robot. This type of robot does not take adjacency of locations into account, making it able to fly between any two locations regardless of their proximity. The flying_robot still has a carrier, and the only action that is different from the normal box_robot is that the movement now happens with the `fly_carrier` action, such that the `adjacent` predicate is not taken as a precondition.

Another type of robot we have made is the shilded_bot, which is a type of earth_robot, which is a type of box_robot. The normal box_robot is now a non_shilded_bot. The idea behind this type of robot is that there are now two types of location within the hospital, normal_location and dangerous_location (e.g. an area with high radiation). The shilded_bot can move through both of these types, while the non_shilded_bot can only move through normal_locations. Also the shilded_bot has a carrier like the other types. Its movement actions, which are only used while moving through a dangerous_location, are `move_carrier_shiled` and `move_shield`. With these dangerous_locations we make the assumption that a person and guide_robot cannot move through these locations, so people can only access units/locations which have a path containing no dangerous_location from the entrance.

V. Problem 3

The third problem uses the same scenario presented in Section IV but instead of using classical planning, we have to develop the domain with Hierarchical Task Networks (HTN).

The first thing was to transform the first domain to from a classical planning one to a HTN one. In order to do this, we first had to identify whether two or more actions could be grouped together to create a task. We identified 3 groups of tasks:

- *Motion* tasks: All the task related to the motion part.
- *Box* tasks: All the task related to the supply delivery part.
- *Accompany* tasks: All the task related to the accompanying part.

Then the tasks are decomposed into subtask from a high level description to a low level description (reaching the *action granularity*). This process exploit the methods, these are a construct of HTN used for decomposing a task into subtasks. What follows now is a complete description of all the tasks.

A. Motion tasks

In this part, we had to transform the motion action into tasks. We have identified 3 different tasks for this purpose: the `move_task`, `move_carrier_task` and the `flying_task`. We decomposed the motion as a recursive task since if the location that the robot has to reach is adjacent to the one in which is located the task is decomposed only in a *move* action, instead if the location is not adjacent the task is decomposed into a *move* action plus another motion task. The same thing was done for the flying task and also the move carrier task.

B. Box tasks

For this part, we identified three different tasks: `complete_order_task`, `load_supply_task`, and `unload_supply_task`. The first task specifies the unit to which the supply must be delivered and is decomposed into the other two tasks. The *load supply* task, on the other hand, guides the robot to the supply and loads it into the carrier. This task is further decomposed into a *go to carrier task* and a *load carrier* action. The last task, which complements the previous one, is used for delivering the supply to the unit. It is decomposed into a *move carrier task* and an *unload carrier* action.

C. Accompany tasks

For this group, we identified three other tasks: `guide_task`, `start_accompany_task`, and `accompany_task`. The first task specifies the unit to which the person must go and is decomposed into the other two tasks. The second task initiates the process of accompanying a person and is further decomposed into a *go to task* and a *start accompany* action. The last task involves accompanying a person to a unit and is decomposed into a *go to task* and an *accompany* action.

VI. Problem 4

The forth problem uses the same scenario presented in Section IV, but instead of using classical planning, the domain must be developed using a temporal planning approach. To enable this, it is necessary to add `:durative-actions` to the requirements section and use a planner that supports this feature (e.g., *OPTIC*).

The creation of the scenario is straightforward and involves changing all `action` constructs into `durative-action` constructs. The main difference is that the preconditions are now divided into three categories: *at start* preconditions, *over all* preconditions, and *at end* preconditions. Similarly, the effects are categorized into *at start* effects and *at end* effects. Additionally, the duration of each action must be explicitly specified.

This new domain allows for parallel actions within the plan, provided the actions do not conflict over shared resources.

To illustrate the process, let us present an example of transforming an action into a durative action to provide an intuitive understanding. The action in Listing 1 represents the `move_carrier` action developed in Section IV. To modify it into a durative action, the preconditions must be divided. We identified that only at the start of the action do the robot's location and the carrier's location need to be at the starting position. Throughout the duration of the action, the two locations must remain adjacent, and the robot must hold the carrier.

For the effects, we specify that the *at end* effects are that the robot and the carrier reach the final location, while the *at start* effects indicate that the robot and the carrier are no longer at the previous location.

A similar approach is applied when creating the remaining actions.

Listing 1. PDDL move_carrier action.

```
(:action move_carrier
  :parameters (
    ?r - box_robot
    ?from - location
    ?to - normal_location
    ?c - carrier
  )
  :precondition (and
     (atl ?r ?from)
     (adjacent ?from ?to)
     (has ?r ?c)
     (at_location_carrier ?c ?from)
  )
  :effect (and
    (not (atl ?r ?from))
    (not(at_location_carrier ?c ?from))
    (atl ?r ?to)
    (at_location_carrier ?c ?to)
  )
)
```

The complete action definition could be found on the GitHub repository, now a brief explanation is done for each action:

- `fly_carrier`: Duration 1. Preconditions: At start, the robot and carrier are at the starting location, and the robot has the carrier. Effect: At start, they are no longer at the starting location; at end, they are at the destination.
- `move_carrier`: Duration 1. Preconditions: At start, the robot and carrier are at the starting location, and the robot has the carrier; the locations are adjacent. Effect: At start, they are no longer at the starting location; at end, they are at the destination.
- `move_carrier_shield`: Duration 2. Preconditions: Same as `move_carrier`, but the locations are dangerous. Effect: Same as `move_carrier`.
- `move`: Duration 1. Preconditions: At start, the robot is at the starting location, and the locations are adjacent. Effect: At start, the robot is no longer at the starting location; at end, it is at the destination.
- `move_shield`: Duration 2. Preconditions: Same as `move`, but the locations are dangerous. Effect: Same as `move`.
- `load_carrier`: Duration 3. Preconditions: At start, the box is empty, the robot is at the location, and the carrier

is at the location; the robot has the carrier, and the supply is at the location. Effect: At start, the box is no longer at the location, and the slot is no longer empty; at end, the box holds the supply, and the slot contains the box.

- `unload_carrier`: Duration 3. Preconditions: At start, the box holds the supply and the slot contains the box; the robot is at the location, and the unit belongs there. Effect: At start, the box no longer holds the supply, and the slot is empty; at end, the box is empty, and the supply is at the unit.
- `start_accompany`: Duration 1. Preconditions: At start, the person is at the location, and the robot is at the location. Effect: At start, the person is no longer at the location; at end, the robot is accompanying the person.
- `accompany`: Duration 1. Preconditions: At start, the robot is accompanying the person; the robot is at the location, and the unit belongs there. Effect: At start, the robot is no longer accompanying the person; at end, the person is in the unit, and at the location.

VII. Problem 5

The fifth problem builds upon the same scenario presented in Section VI, implementing a *PlanSys2* infrastructure to integrate the scenario into a ROS2 environment. The first step involves setting up the environment and creating an *ActionExecutorClient* for each action defined in the domain. These will be placeholder actions, as the executor will simply print the status of each action without performing any additional tasks.

Next, a launch file must be created to initiate the domain expert, all the action executors, and an instance of the *PlanSys2 terminal*. In the terminal, the problem (in the correct format) can be loaded, and the planner can be run to find a solution. Once a solution is found, the `run` command can be used in the terminal to execute the plan. In this case, the execution will consist only of status prints. However, if specific behaviors were assigned during the action definition, the associated behavior would be executed during this phase.

During the development of the *PlanSys2* infrastructure, we encountered a significant issue. When instantiating the problem, certain facts, specifically those related to robots, were not recognized by the domain expert. This initially led us to believe there was an issue with our domain file. However, after some debugging, we discovered that the problem was not on our end but rather a limitation of *PlanSys2*. It turns out that *PlanSys2* supports inheritance from at most two types.

For instance, the type `shielded_bot` is not recognized because it inherits from `earth_robot`, which inherits from `box_robot`, which in turn inherits from `robot`, ultimately inheriting from the `object` type. Conversely, if a `box_robot` is used, the issue does not occur.

To address this issue, we limited our domain to using only `box_robot` and `guide_robot`, avoiding the use of additional robot types. This small change simplified the domain structure and ensured compatibility with *PlanSys2*'s inheritance constraints. By doing so, we were able to proceed without further issues related to type recognition.

VIII. Evaluation of the problems

For evaluating the problems, we adopted an *increasing complexity* strategy. So we will start evaluating with simple cases till more difficult ones. For this sake we have created two different maps, that have different complexities. We have

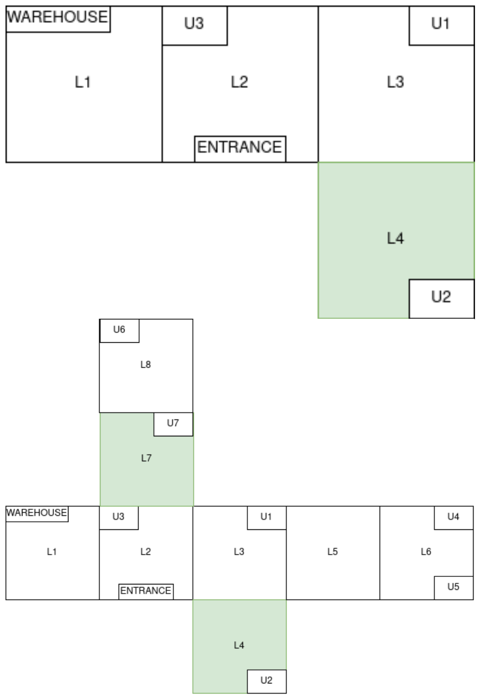


Fig. 1. The upper one is the simplest map, the lower one is the one that we have used for final test.

created 4 different problem instances for each problem, they are the same problems applied to the different domains.

1) First one is on the simplest map with only one delivery task;
2) Second one is on the simplest map with one delivery task and one accompany task;
3) Third one is one the simplest map with 3 delivery tasks and 2 accompany tasks;
4) Forth one is on the hardest map with 5 delivery tasks and 4 accompany tasks

The evaluation will be done considering the total time that the solver needs to find a plan, the path length and the expanded nodes.

A. Problem 1

For problem 1, we have made a comarison between three planners: Fast Forward, Fast Downward, and Scorpion. In addition, we also made a comparison on the performance between the domain with supplies as objects and the alternative domain where supplies were modeled as predicates and actions. Throughout our tests we have used four problem files, the first two of which are more for testing purposes. To the end of reducing redundancy, we have chosen to not show results for these first two problems, instead focusing on comparing results of problem 3 and 4, described above.

For the results, which can be seen in Tables VIII-A, VIII-A, we have used the following commands within the planutils Docker image.

```
ff domain.pddl problem<1-4>.pddl

downward --alias seq-sat-fdss-2023
--overall-time-limit 60s
domain.pddl problem<1-4>.pddl

scorpion --alias seq-sat-fdss-2023
--overall-time-limit 60s
domain.pddl problem<1-4>.pddl
```

Originally, we were simply using downward without an alias and with the search heuristic `astar(lmcut())`. However, for problem4, this led to a runtime of over 10 minutes, after which we halted the search, as no results were coming in. This led us to research some possibilities to limit the runtime. First, we tested with different search heuristics to see if the search time would go down. Examples of these are:

```
downward domain.pddl problem<1-4>.pddl
--search "astar(blind())"
```

and

```
downward domain.pddl problem<1-4>.pddl
--evaluator "hff=ff()"
--search "lazy_greedy([hff])"
```

This did not seem to help decrease the runtime to somewhere around 60 seconds, which is why we shifted our focus to being able to stop the search after some time and still get a (sub)optimal plan. We came across the `overall-time-limit` flag, which did not work on its own, as now it would simply abort after a minute without outputting a plan. Then by testing several aliases, we came across one that worked with a timeout.

For our comparison we have taken a look at the runtime of the planner, the length of the path it generated, and the number of states (or peak memory usage, if the number has KB behind it). From these values we can observe the following.

Both in problem 3 and 4, the ff planner generates a path much faster, but always using more actions than its counterparts, meaninging it is suboptimal. We can also see that the length of the path that ff generates is smaller, and closer to those of downward/scorpion, when using supplies as predicates. With problem 3, the peak memory usage is lower for supplies as predicates, whereas in problem 4 it is slightly higher. This is unexpected, as we hypothesized that the search space would be larger with supplies as predicates. A possible explanation could be that in problem 3, the supplies as objects case comes to the path length 20 solution faster, and thus has time to go into a deeper, more intensive search, which leads to higher memory usage. But this is not a claim grounded on deeper analysis of the planners. Finally, we can also see that the memory usage for scorpion is somewhat lower than that of downward, even though scorpion is based on downward. This suggests some optimizations done within the scorpion planner.

Looking at this data, one of the strongest conclusions, or at least points for further exploration, is the significant decrease in path length for the ff planner between using supplies as objects or predicates/actions. As we have discussed before, within an industry setting, this is not a very practical approach, but it could still be interesting to look further into expanding domains and removing objects to improve the quality of the ff planner.

Planner	Version	Time (s)	Length	States/ Memory
ff	Problem 3: supplies as objects	0.0	24	191
	Problem 3: supplies as predicates	0.0	22	195
downward	Problem 3: supplies as objects	60.0	20	60644KB
	Problem 3: supplies as predicates	60.0	20	48288KB
scorpion	Problem 3: supplies as objects	60.0	20	55020KB
	Problem 3: supplies as predicates	60.0	20	41224KB

Planner	Version	Time (s)	Length	States/ Memory
ff	Problem 4: supplies as objects	0.02	54	1787
	Problem 4: supplies as predicates	0.01	44	958
downward	Problem 4: supplies as objects	60.0	42	38964KB
	Problem 4: supplies as predicates	60.0	42	40252KB
scorpion	Problem 4: supplies as objects	60.0	42	34732KB
	Problem 4: supplies as predicates	60.0	42	35872KB

B. Problem 2

For problem 2 we decided to drop the comparison with supplies as predicates and solely focus on supplies as objects, for the reasons stated in Section III-D. We have used the same commands as in part 1 of the project. Looking at the results in Table VIII-B, similar conclusions can be drawn about the comparison of the different planners as in part 1. Notable is that the introduction of other vehicles has led to significantly lower path lengths. We suspect this can be attributed to the flying_robot, which can instantly travel between locations.

Planner	Version	Time (s)	Path Length	States/ Peak Memory
ff	Problem 3	0.0	21	229
	Problem 4	0.01	37	721
downward	Problem 3	60.0	19	55504KB
	Problem 4	60.0	33	30976KB
scorpion	Problem 3	60.0	19	51052KB
	Problem 4	60.0	33	27844KB

C. Problem 3

For problem 3 the only planner available was `PANDA` so the benchmark was done only using it. As it can be seen, even with medium-sized problems, the planning time remains reasonable. All the experiments were run with the command:

```
java -jar PANDA.jar -parser hddl
domain.hddl problem<1-4>.hddl
```

Version	Total time (s)	Path length	States
Problem 1	0.837	5	36
Problem 2	0.925	8	52
Problem 3	1.004	22	230
Problem 4	1.441	42	587

D. Problem 4

For problem 4 the planner that we used for evaluating our scenarios are `Optic` and `Temporal Fast Downward`, We have selected these since they support `durative-actions` requirement. In this part we will display the results only for the instance 3 and the instance 4 just to not have redundant information in the report.

An important sidenote to do is that these two planners were runned inside a `Docker` environment.

Planner	Version	Total time (s)	Path length	States
Optic	Problem 3	47.11	16	46
	Problem 4	132.91	37	183
TFD	Problem 3	0.278	18	59
	Problem 4	0.536	38	259

As we can see now `TFD` has better performances compared to `Optic`. This behavior was seen also in class, so it is to be expected. This is caused by the different strategies and heuristics that the two planner adopt. Despite this, both planners are able to find a solution.

All the experiments were run with the command:

```
tfd domain.pddl problem<1-4>.pddl
optic domain.pddl problem<1-4>.pddl
```

E. Problem 5

The problem five was evaluated using only the harder problem instance. Now the domain is a modified version of the original one since the inheritance issue of `PlanSys2` that we have discussed in Section VII.

The planner used by default is `POPF`, we didn't saw any particular difference in terms of Total time between this planner and `TFD`.

As for the previous cases, the planner is able to solve the problem and the `PlanSys2` infrastructure is able to run the plan using the *fake actions* we have designed.

For running this experiment the commands used are:

```
ros2 launch problem_5 start.launch.py
```

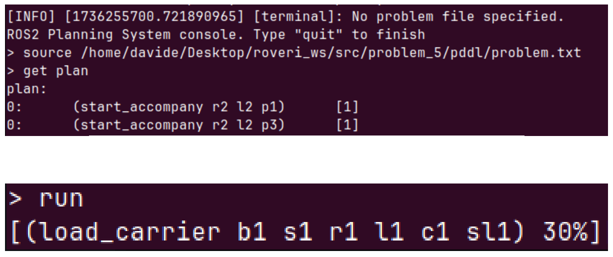
Then in another terminal run the PlanSys2 terminal

```
ros2 run plansys2_terminal plansys2_terminal
```

In the terminal source the problem file

```
> source /path/to/problem
```

Then retrieve the plan with the command `get_plan` or directly run `run` for getting the plan and executing it.



```
[INFO] [1736255700.721890965] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
> source /home/davide/Desktop/roveri_ws/src/problem_5/pddl/problem.txt
> get plan
plan:
0:      (start_accompany r2 l2 p1)      [1]
0:      (start_accompany r2 l2 p3)      [1]
```

```
> run
[(load_carrier b1 s1 r1 l1 c1 sl1) 30%]
```

Fig. 2. PlanSys2 terminal instance in which it can be seen that the plan is retrived and executed.

IX. Discussion and Conclusions

While working on this project, we explored various forms of planning and experimented with multiple planners. Ultimately, we built a ROS2 infrastructure to implement and execute our problem in a robotic environment.

During the development process, we encountered several challenges, ranging from the lack of a debugger to more substantial issues, such as inheritance problems with PlanSys2.

One of the most notable and surprising observation is the significant difference in performance between OPTIC and TFD. This difference is substantial and noteworthy, highlighting a clear gap in their efficiency and capabilities under the same conditions. We believe this difference is likely due to the use of different heuristics, with OPTIC potentially focusing more on optimizing the solution compared to TFD.

Another notable observation is the difference in plan length generated by ff with the comparison of the first domain using supplies as objects or as predicates/actions. A possible explanation for this is that the search space actually shrinks when fewer objects are needed, so that ff can come to a more optimal plan more easily. Exploring the actual cause for this, and the extent to which this can be utilized in practice, is something to be researched in the future. What can be said, as has been done earlier in the report, is that the implementation of supplies as predicates/actions is both more intensive on the developer, and makes for a more rigid application, as the supplies are now bound by the domain, and can no longer vary on a problem scale, which would be desirable if this were to be used by different hospitals with different requirements. To this end, we conclude that this alternative implementation does not comply to industrial standards as of yet, and needs to be investigated further to show its potential advantages.