Planning Systems in ROS2

Carmine Tommaso Recchiuto

PDDL

PDDL stands for Planning Domain Definition, and it is an attempt of defining a standard encoding language for “classical” planning tasks.

The main components of a PDDL planning tasks are:

Objects -> Things in the world that interest us.

Predicates -> Properties of objects that we are interested in; can be *true* or *false*

Initial state -> The state of the world that we start in.

Goal specification -> Things that we want to be true

Actions / Operators -> Ways of changing the state of the world

PDDL

Usually all these elemens are separated in two parts:

- A **domain file** for objects, predicates and actions: these may be reused for different problem files

- A **problem file** for objects, initial state, and goal specification.

PDDL

Usually all these elemens are separated in two parts:

- A **domain file** for objects, predicates and actions: these may be reused for different problem files

- A **problem file** for objects, initial state, and goal specification.

A problem file describes one specific instance that can occur in the described domain. A domain file defines the “universal” aspects of a problem. Essentially, these are the aspects that do not change regardless of what specific situation we’re trying to solve.

Domain

```
(define (domain <domain name>)
(:predicates
  <predicate-list>
)

(:action
  <action-details>
)
)
```

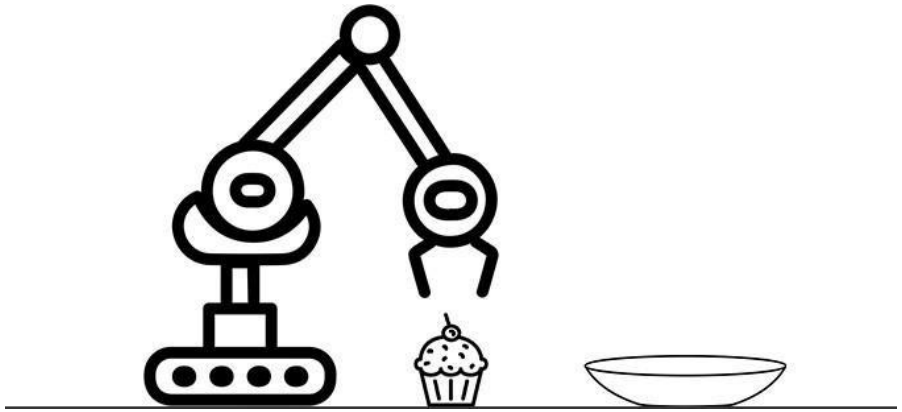
```
(define (problem <title>)
                (:domain <domain-name>)
                (:objects
                <object-list>
                )

                (:init
                                <predicates>
                )
                (:goal
                                <predicates>
                )
)
```

Problem

Simple Example: Let's Eat

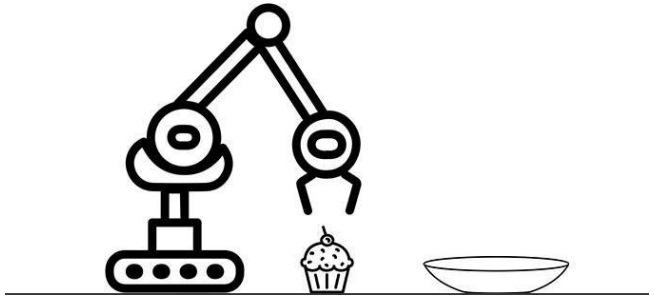
Let's imagine to have a robot gripper arm, a cupcake and a plate. The gripper is empty, the cupcake is on the table, and we want to put the cupcake on the plate.



If we want to model this situation in PDDL, we need at first to define the domain.

(define (domain robot_eat))

Simple Example: Domain



We need than to model all objects that we may have in our domain, …

```
(:types
   robot
   object
   location
)
```

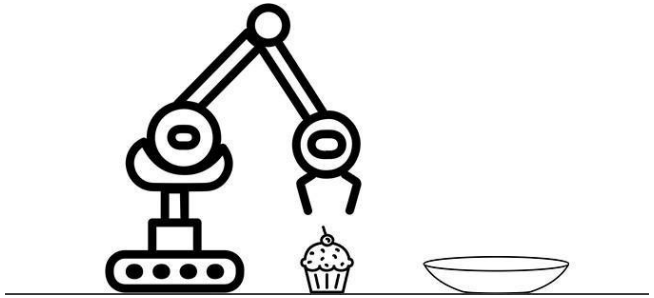
…, the predicates (true/false conditions that may be applied to the objects of our domain, …

```
(:predicates
   (object_on ?object - object  ?loc - location)
   (robot_on ?robot - robot ?loc - location)
   (holding ?robot - robot ?object - object)
   (empty ?robot - robot)
)
```

Simple Example: Domain

… and the actions/operators, that can change the state of the world:

```
(:action pick-up
:parameters (?robot - robot ?object - object ?loc - location)
:precondition (and
    (robot_on ?robot ?loc)
    (object_on ?object ?loc)
    (empty ?robot)
    )
:effect (and
    (not (object_on ?object ?loc))
    (not (empty ?robot))
    (holding ?robot ?object))
)
```



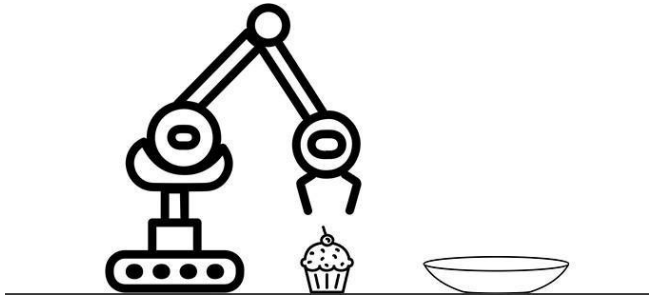
The **parameters** defines the type of object we're interested in

Preconditions are conditions which must be met in order for an action to be possible.

An **effect** consists of a conjunctive logical expression, which defines which values should be set to true or false if an action is applied.

Simple Example: Domain

```
(:action drop
:parameters (?robot - robot ?object - object ?loc - location)
:precondition (and
    (robot_on ?robot ?loc)
    (holding ?robot ?object)
)
:effect (and
    (object_on ?object ?loc)
    (empty ?robot)
    (not (holding ?robot ?object))
)
)
```



We have our domain file!

```
(:action move
:parameters (?robot - robot ?from - location ?to - location)
:precondition (robot_on ?robot ?from)
:effect (and
    (not (robot_on ?robot ?from ))
    (robot_on ?robot ?to)
)
)
```

Simple Example: Requirements

We only need to add to our domain file the **requirements.**

Requirements are similar to import/include statements in programming languages.

E.g.

Strips -> Allows the usage of basic add and delete effects as specified in STRIPS. e.g.:effect (empty ?robot – gripper)

Typing -> Allows the usage of typing for objects. Typing is similar to classes and sub-classes in Object Oriented Programming

(:types
site material - object
bricks cables windows - material
)

Disjunctive Preconditions -> Allows the usage of or in preconditions

Durative Actions -> Durative actions are actions which have a duration they take to complete. In this case, effects and conditions may be specified *at start, at end, over all,* and actions may also have a duration assigned.

Simple Example: Problem

Dealing with the Problem file, we should associate a domain to it, and define the objects that exist in its world.

```
(define (problem letseat-simple)
        (:domain robot_eat)
        (:objects
        arm - robot
        cupcake - object
        table - location
        plate - location)
```

Then, we'll define the initial state: the gripper is empty, the cupcake and the robot are on the table.

```
(:init
        (robot_on arm table)
        (object_on cupcake table)
        (empty arm)
)
```

And finally, we define the **goal**

```
(:goal
        (object_on cupcake plate)
)
```

Simple Example: Planner

Given the domain and the corresponding problem, the planner will generate something like:

0: (pick-up arm cupcake table)
1: (move arm table plate)
2: (drop arm cupcake plate)

This is our plan!

It would be quite useful having the possibility of using a framework that generates plan (and possibly also problems), uses the generated plan for directly controlling the robot, and also checks that actions have been accomplished without errors.

PlanSys!

ROS2 – Plansys2

PDDL-based planning system for ROS2, strongly inspired by ROSPlan, but with some additional features.

You can directly install it with apt install:

(sudo) apt install ros-jazzy-plansys2-*

ROS2 – Plansys2

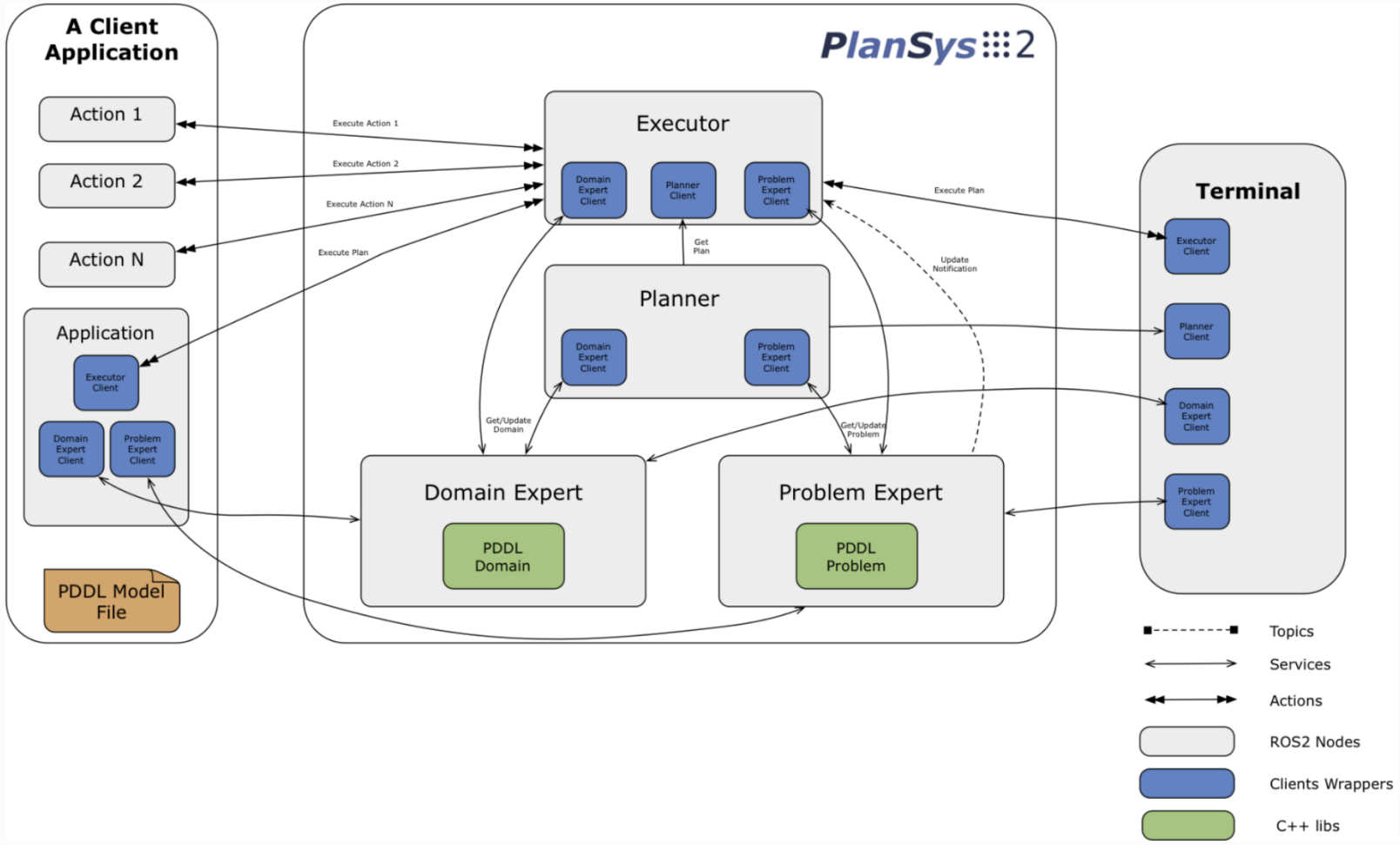
PlanSys2 has a modular design, with 4 nodes:

Domain Expert: Contains the PDDL model information

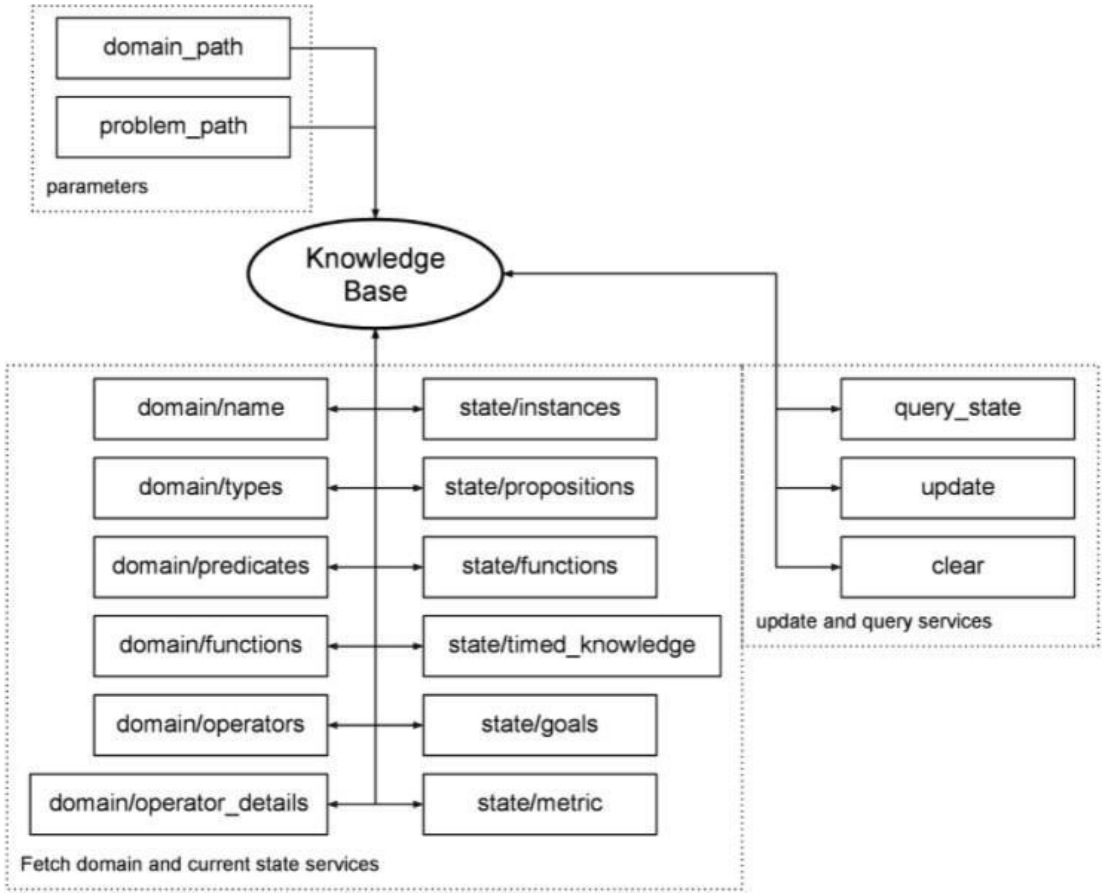
Problem Expert: Contains the current instances, predicates, functions, and goals

Planner: Generates plans using the information contained in the Domain and Problem Experts.

Executor: Takes a plan and executes it by activating the action performers (the nodes that implement each action).



Domain Expert



(ROS1 version)

The Domain Expert is used to store a PDDL model. In particular, it stores both a domain model and the current problem instance. Practically, it:

- loads a PDDL domain (and optionally a problem) from a file

- stores the state as a PDDL instance

- may be updated by ROS services.

- can be queried

Domain Expert

Parameters

model_file [string: PDDL model files to load (it can be even more than one, for modular applications).

Topic

domain_expert/domain [std_msgs::msg::String]: A message is published on this topic with the current domain.

Services

•domain_expert/get_domain_name [plansys2_msgs::srv::GetDomainName]: Get the name of the domain.

•domain_expert/get_domain_types [plansys2_msgs::srv::GetDomainTypes]: Get the valid types.

•domain_expert/get_domain_actions [plansys2_msgs::srv::GetDomainActions]: Get the available actions.

•domain_expert/get_domain_action_details [plansys2_msgs::srv::GetDomainActionDetails]: Get the details of a specific action.

•domain_expert/get_domain_durative_actions [plansys2_msgs::srv::GetDomainDurativeActions]: Get the available durative actions.

•domain_expert/get_domain_durative_action_details [plansys2_msgs::srv::GetDomainDurativeActionDetails]: Get the details of a specific durative action.

•domain_expert/get_domain_predicates [plansys2_msgs::srv::GetDomainPredicates]: Get the valid predicates.

•domain_expert/get_domain_predicate_details [plansys2_msgs::srv::GetDomainPredicateDetails]: Get the details of a specific predicate.

•domain_expert/get_domain_functions [plansys2_msgs::srv::GetDomainFunctions]: Get the valid functions.

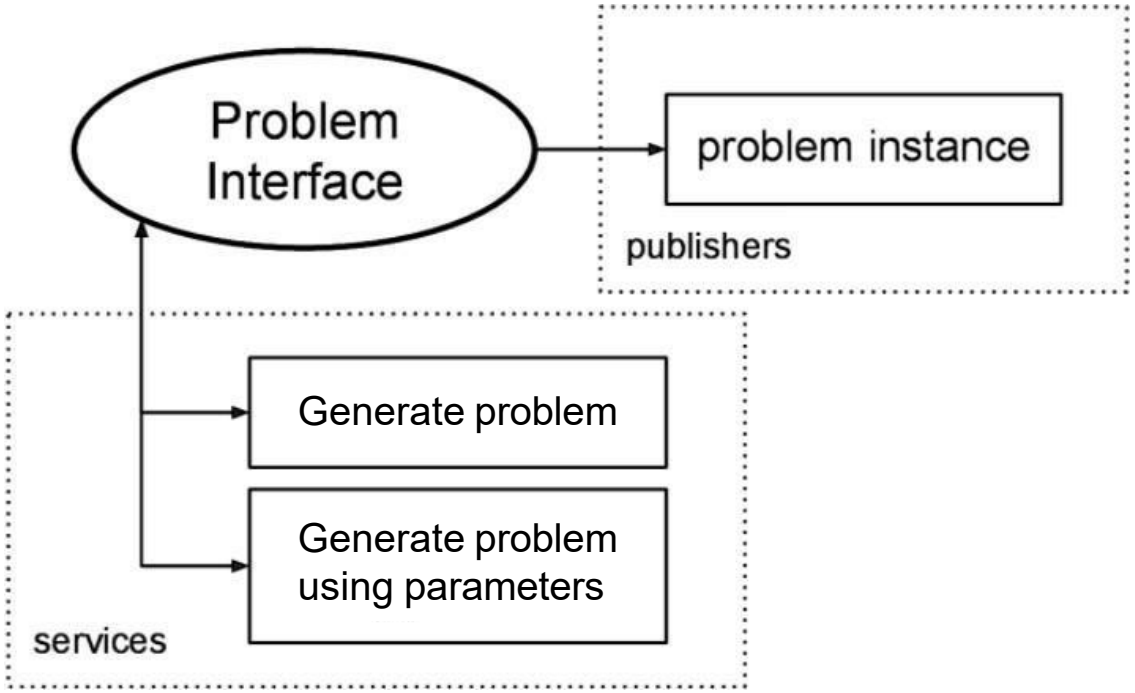
•domain_expert/get_domain_function_details [plansys2_msgs::srv::GetDomainFunctionDetails]: Get the details of a specific function.

•domain_expert/get_domain_derived_predicates [plansys2_msgs::srv::GetDomainDerivedPredicates]: Get the valid derived predicates.

•domain_expert/get_domain_derived_predicate_details [plansys2_msgs::srv::GetDomainDerivedPredicateDetails]: Get the details of a specific derived predicate.

•domain_expert/get_domain [plansys2_msgs::srv::GetDomain]: Get the PDDL domain as a string.

Problem Expert



(ROS1 version)

The Problem Expert node is used to generate a proble instance.

It fetches the domain details and current state through service calls to the Knowledge Base node, and publishes a PDDL problem instance as a string, also writing it to a file.

Problem Expert

Parameters

model_file [string: PDDL model files to load (it can be even more than one, for modular applications).

Publishers / Subscriber

•problem_expert/problem [plansys2_msgs::msg::Problem]: A message is published on this topic with the current problem.

•problem_expert/update_notify [std_msgs::msg::Empty] A message is published on this topic when any element of the problem changes.

•problem_expert/knowledge [plansys2_msgs::msg::Knowledge] A message is published on this topic when any element of the problem changes.

Services (not exhaustive)

•problem_expert/add_problem [plansys2_msgs::srv::AddProblem]: Add a problem to the current knowledge.

•problem_expert/add_problem_goal [plansys2_msgs::srv::AddProblemGoal]: Replace the goal.

•problem_expert/add_problem_instance [plansys2_msgs::srv::AffectParam]: Add an instance.

•problem_expert/add_problem_predicate [plansys2_msgs::srv::AffectNode]: Add a predicate.

•problem_expert/add_problem_function [plansys2_msgs::srv::AffectNode]: Add a function.

•problem_expert/get_problem_goal [plansys2_msgs::srv::GetProblemGoal]: Get the current goal.

•problem_expert/get_problem_instance [plansys2_msgs::srv::GetProblemInstanceDetails]: Get the details of an instance.

•problem_expert/get_problem_instances [plansys2_msgs::srv::GetProblemInstances]: Get all the instances.

•problem_expert/get_problem_predicate [plansys2_msgs::srv::GetNodeDetails]: Get the details of a predicate.

•problem_expert/get_problem_predicates [plansys2_msgs::srv::GetStates]: Get all the predicates.

•problem_expert/get_problem_function [plansys2_msgs::srv::GetNodeDetails]: Get the details of a function.

•problem_expert/get_problem_functions [plansys2_msgs::srv::GetStates]: Get all the functions.

•problem_expert/get_problem [plansys2_msgs::srv::GetProblem]: Get the PDDL problem as a string.

•problem_expert/remove_problem_goal [plansys2_msgs::srv::RemoveProblemGoal]: Remove the current goal.

•problem_expert/remove_problem_instance [plansys2_msgs::srv::AffectParam]: Remove an instance.

•problem_expert/remove_problem_predicate [plansys2_msgs::srv::AffectNode]: Remove a predicate.

•problem_expert/remove_problem_function [plansys2_msgs::srv::AffectNode]: Remove a function.

Planner

This component calculates the plan to obtain a goal.

Each PDDL solver in PlanSys2 is a plugin. By default PlanSys2 uses POPF, although other PDDL solvers can be used easily. Currently, the Temporal Fast Downward (TFD) solver is also available.

A plan is usually requested by providing a domain acquired from the Domain Expert and a problem acquired from the Problem expert.

Also in this case, we have some parameters and services that we can use:

Parameters
plan_solver_plugins [vector<string>]: List of PDDL solver plugins

Services (not exhaustive!)

planner/get_plan [plansys2_msgs::srv::GetPlan]: Get a plan that will satisfy the provided domain and problem..
planner/validate_domain [plansys2_msgs::srv::ValidateDomain]: Validate the provided domain.

Example

First, we have to define the domain. You can refer to the ROS2 package:
https://github.com/CarmineD8/plansys_interface

Let's give a look to the domain: example.pddl

```
define (domain simple)
(:requirements :strips :typing :adl :fluents :durative-actions)
```

:fluents -> Allows the inclusion of a :function block which represent numeric variables in the domain. e.g.

:adl -> adds the following requirements
:disjunctive-preconditions
:equality
:quantified-preconditions
:conditional-effects

You can give a look here to know more about the requirements that can be needed for the definition of the domain:
https://planning.wiki/ref/pddl/requirements

Example

```
;; Types ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:types
robot
room
);; end Types ;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;
(:predicates

(robot_at ?r - robot ?ro - room)
(connected ?ro1 ?ro2 - room)
(battery_full ?r - robot)
(battery_low ?r - robot)
(charging_point_at ?ro - room)

);; end Predicates ;;;;;;;;;;;;;;;;;;;;;;
```

In the domain, we define two types (robot, room)

5 predicates, which describe relations and attributes of the types

Example

```
;; Actions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:durative-action move
    :parameters (?r - robot ?r1 ?r2 - room)
    :duration ( = ?duration 5)
    :condition (and
        (at start(connected ?r1 ?r2))
        (at start(robot_at ?r ?r1))
        (over all(battery_full ?r))
        )
    :effect (and
        (at start(not(robot_at ?r ?r1)))
        (at end(robot_at ?r ?r2))
    )
)
```

```
(:durative-action askcharge
    :parameters (?r - robot ?r1 ?r2 - room)
    :duration ( = ?duration 5)
    :condition (and
        (at start(robot_at ?r ?r1))
        (at start(charging_point_at ?r2))
       )
    :effect (and
        (at start(not(robot_at ?r ?r1)))
        (at end(robot_at ?r ?r2))
    )
)
```

Example

```
(:durative-action charge
    :parameters (?r - robot ?ro - room)
    :duration ( = ?duration 5)
    :condition (and
        (at start(robot_at ?r ?ro))
        (at start(charging_point_at ?ro))
    )
    :effect (and
         (at end(not(battery_low ?r)))
         (at end(battery_full ?r))
    )
)

);; end Domain ;;;;;;;;;;;;;;;;;;;;;;;;
```

And 3 actions for managing the motion of our robot in the environment.

Problem

…We also need a problem…

In this example, the robot has initially the battery at a low level. It is in the sitting_room, while the charging point is in the corridor.

The final goal is to reach the bathroom!

```
( define ( problem problem_1 )
( :domain simple )
( :objects
                robot1  - robot
  bedroom kitchen bathroom sitting_room corridor - room
)
( :init
                ( robot_at robot1 sitting_room )

  ( connected sitting_room corridor )
  ( connected corridor sitting_room )
  ( connected bedroom corridor )
  ( connected corridor bedroom )
  ( connected bedroom bathroom )
  ( connected bathroom bedroom )
  ( connected sitting_room kitchen )
  ( connected kitchen sitting_room )

  ( charging_point_at corridor)
  ( battery_low robot1)
)
( :goal
                ( and
                                ( robot_at robot1 bathroom)
                ))
)
```

Launch

…and a launch file, to start all required nodes…

PlanSys2 can be launched with two different launchers that implement two different execution forms: distributed or monolithic. In distributed form, each component of PlanSys2 runs in a different process, and to launch it, each component's launchers are called in cascade. In monolithic form, all components are released in the same process.

We are going to use the distributed version, which is more easy to "control".

In the launch file (distributed.launch.py) we start four nodes: the domain, the planner expert and the planner (the ones that we have already analyzed), plus the lifecycle manager, which is needed for correctly managing all the plansys nodes.

There are now two different strategies to interact with the knowledge base: services, or terminal.

Services

With services, we can interact with PLANSYS by:

- Querying the domain_expert
- Querying and modifying the problem_expert
- Trigger the planner to generate the plan.

If you have already defined the problem in a file (as in our case), you may directly trigger the planner. Still, you need first to retrieve the current domain and problem (as strings).

This can be done in a smooth way in a cpp code (see getplan.cpp)

GetPlan.cpp

```
void init()
 {
   domain_expert_ = std::make_shared<plansys2::DomainExpertClient>();
   planner_client_ = std::make_shared<plansys2::PlannerClient>();
   problem_expert_ = std::make_shared<plansys2::ProblemExpertClient>();
 }

 void plan()
 {

        auto domain = domain_expert_->getDomain();
        auto problem = problem_expert_->getProblem();
        auto plan = planner_client_->getPlan(domain, problem);

        if (!plan.has_value()) {
          std::cout << "Could not find plan to reach goal " <<
            parser::pddl::toString(problem_expert_->getGoal()) << std::endl;
        }

        else{
        std::cout << plan.value() << std::endl;
      }
 }
```

Plansys2 Terminal

An additional feature of Plansys2 is the possibility of using a terminal to interact with the framework

ros2 run plansys2_terminal plansys2_terminal

```
get domain

( define ( domain simple )
( :requirements :strips :adl :typing :durative-actions :fluents )
( :types
  robot - object
  room - object
)
( :predicates
  ( robot_at ?robot0 - robot ?room1 - room )
  ( connected ?room0 - room ?room1 - room )
  ( battery_full ?robot0 - robot )

  ...
```

```
get model types
Types: 2
          robot
          room
```

```
get model actions
Actions: 0
  move (durative action)
  askcharge (durative action)
  charge (durative action)

get model predicates
Predicates: 5
  robot_at
  connected
  battery_full
  battery_low
  charging_point_at
```

```
get model predicate robot_at
Parameters: 2
 robot - ?robot0
 room - ?room1

get model action move
Type: durative-action
Parameters: 3
 ?0 - robot
 ?1 - room
 ?2 - room
AtStart requirements: (and (connected ?1 ?2)(robot_at ?0 ?1))
OverAll requirements: (and (battery_full ?0))
AtEnd requirements:
AtStart effect: (and (not (robot_at ?0 ?1)))
AtEnd effect: (and (robot_at ?0 ?2))
```

Plansys2 Terminal

We can use the terminal also to remove (and add) instances (the problem) and generate the plan!

```
set instance leia robot
set instance entrance room
set instance kitchen room
set instance bedroom room
set instance dinning room
set instance bathroom room
set instance chargingroom room
```

```
set predicate (connected entrance dinning)
set predicate (connected dinning entrance)

set predicate (connected dinning kitchen)
set predicate (connected kitchen dinning)

set predicate (connected dinning bedroom)
set predicate (connected bedroom dinning)

set predicate (connected bathroom bedroom)
set predicate (connected bedroom bathroom)

set predicate (connected chargingroom kitchen)
set predicate (connected kitchen chargingroom)

set predicate (charging_point_at chargingroom)
set predicate (battery_low leia)
set predicate (robot_at leia entrance)
```

```
set goal (and(robot_at leia bathroom))
```

The terminal can be used to check if instances have been correctly updated:

```
get problem instances

get problem predicates
```

Plansys2 Terminal

Finally, we can generate the plan!

> get plan (using the terminal)
ros2 run popf popf /tmp/domain.pddl /tmp/problem.pddl (from another shell)

The files domain.pddl and problem.pddl have been created by plansys.

What if we try to execute the plan?

> run

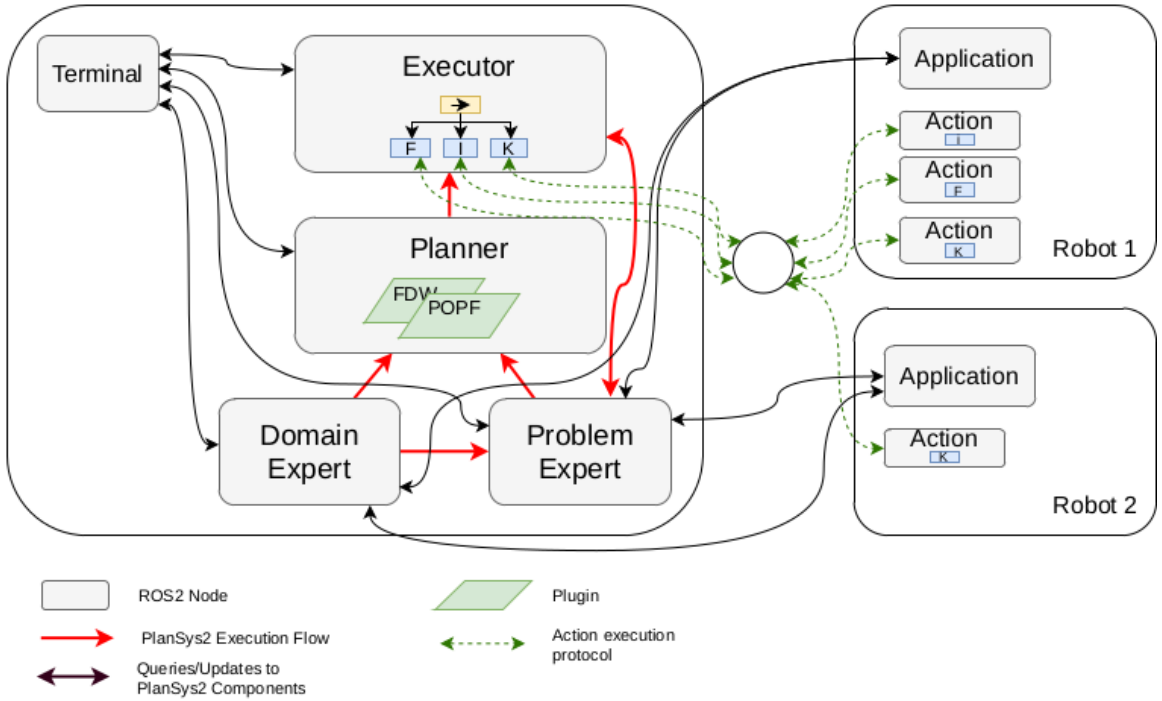
[ERROR] [1764677351.759509695] [executor_client]: Action server not available after waiting
[ERROR] [1764677354.763386840] [executor_client]: send_goal failed
Execution could not start

While we have the four plansys2 nodes running and ready to implement the plan, we also need some actions, on the client side, that are actually able to implement the required actions!

Plansys2 Action Execution

This component is responsible for executing a provided plan. It is, by far, the most complex component since the execution involves activating the action performers. This task is carried out with the following characteristics:

- It optimizes its execution, parallelizing the actions when possible.
- It checks if the requirements are met at runtime.
- It allows more than one action performer for each action, supporting multirobot execution.

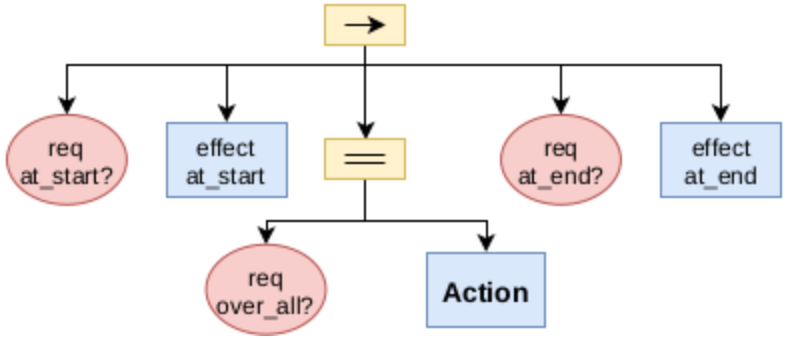


Launch File

We need therefore to add the executor in the launch file: distributed_ex.launch

Please notice that, together with the launch of an additional node (i.e., the executor), we need to define some additional parameters: action_bt_file, start_action_bt_file, end_action_bt_file, bt_builder_plugin

That’s because, once a plan is obtained, the Executor converts it to a Behavior Tree to execute it. Each action becomes the following subtree:



Behaviour Tree

- A behaviour tree is a hierarchical, tree-structured model used to define the behaviour of an agent (robot, AI, system).

- The tree consists of nodes. There are different kinds, but broadly:
  - Control nodes (non-leaf): they define how to combine or choose between children (e.g. sequence, fallback/selector, parallel).
  - Execution (leaf) nodes: actions or conditions. For example “Go to location”, “Pick object”, or “Check sensor” etc.

- Execution works by sending a “tick” from the root, propagating down: each control node determines which child(ren) to tick (or in what order), and leaf nodes run actions or checks. The control nodes aggregate these statuses to decide what to do next.

- This enables modular and flexible design: you can build complex behaviors by combining simpler ones in a tree, reuse subtrees, and manage conditional / sequential / parallel behavior in an organized way.

Behaviour Trees in plansys

In PlanSys2, behaviour trees are used as the *runtime representation* of a plan. Once a plan is generated, it is converted into a behaviour tree that the system executes:

- First, planning computes a sequence of actions, along with dependencies: some actions must happen before others (because of preconditions/effects), some may happen in parallel, etc. PlanSys2 builds a **planning graph** encoding those dependencies.

- Then, from that graph, PlanSys2's **Behavior-Tree Builder** generates a behaviour tree: each action in the plan becomes a subtree (a part of the BT)

- The BT encodes both **ordering** (so that actions with dependencies happen in the right order) and **parallelism** (if independent actions can run in parallel — the BT can schedule them accordingly).

- For each action that is executed via BT, PlanSys2 uses a generic executor node: implementations of actions are provided as BT nodes (leaf or small subtrees).

- PlanSys2 separates **planning** from **execution:** and execution is handled by a dynamically generated behaviour tree that respects the plan logic. This makes the execution also possibly concurrent

Executing the Plan

We can than slightly modify the node for executing the plan, by adding the actual plan execution:

```
executor_client_->start_plan_execution(plan.value());
```

…and a subscriber to monitor plan execution

```
  action_feedback_sub_ = this-
>create_subscription<plansys2_msgs::msg::ActionExecutionInfo>(
        "/action_execution_info", 10,
        std::bind(&Controller::action_feedback_callback, this,
std::placeholders::_1));
```

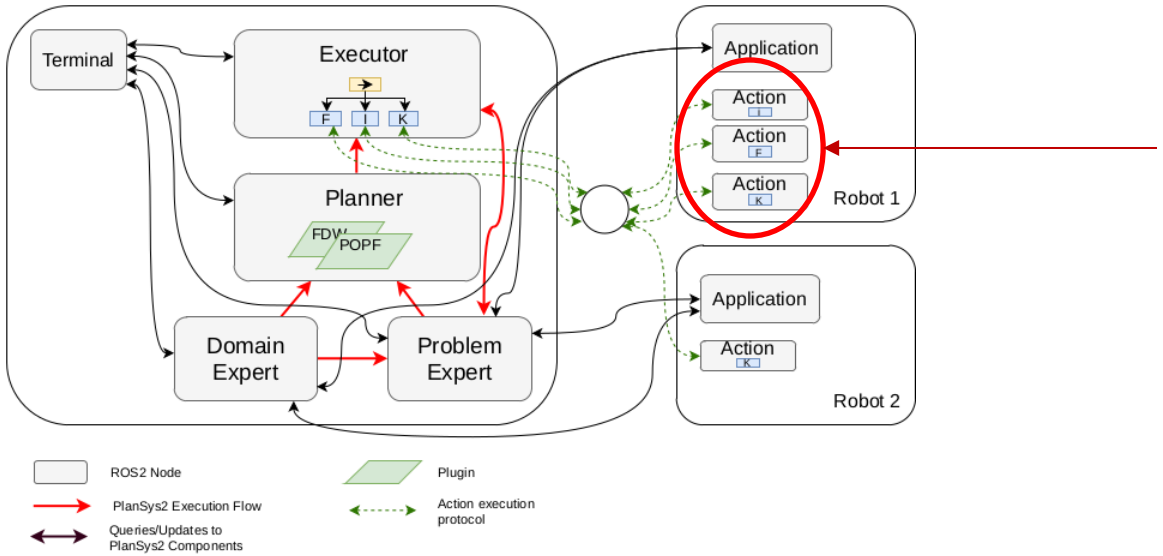
But we can notice that the execution of the plan does not really start!

Behaviour Trees in plansys

Great! If we run now the launch file, and we start the node getplan_and_execute, we obtain this:

[executor_node-4] [WARN] [1764680590.433745102] [executor]: No action performer for (askcharge robot1 sitting_room corridor). Retrying

Please notice that the same result can be obtained with the terminal, by creating the plan (get plan) and typing “run”



We have all the system running now, but we do not have actions that can practically execute the work!

Plansys2 Action Execution

How actions are implemented in Plansys2?

Actions are implemented as subclasses of plansys2::ActionExecutorClient

class MoveAction : public plansys2::ActionExecutorClient

Each action in PlanSys2 is a managed node , also known as lifecycle node. When active, it will iterativelly call a function to perform the job.

Within the constructor, we can set the frequency of the do_work() method, which is the one called when the executor triggers the action.

Here the do_work method only increments a counter (so it's a sort of "simulated action"). Specifically, you can notice two functions used by the action:

send_feedback()
finish()

This code is sending feedback of its completion. When finished, finish method indicates that the action has finished, and send back if it succesfully finished, the completion value in [0-1] and optional info. Then, the node will pass to inactive state.

Launch file

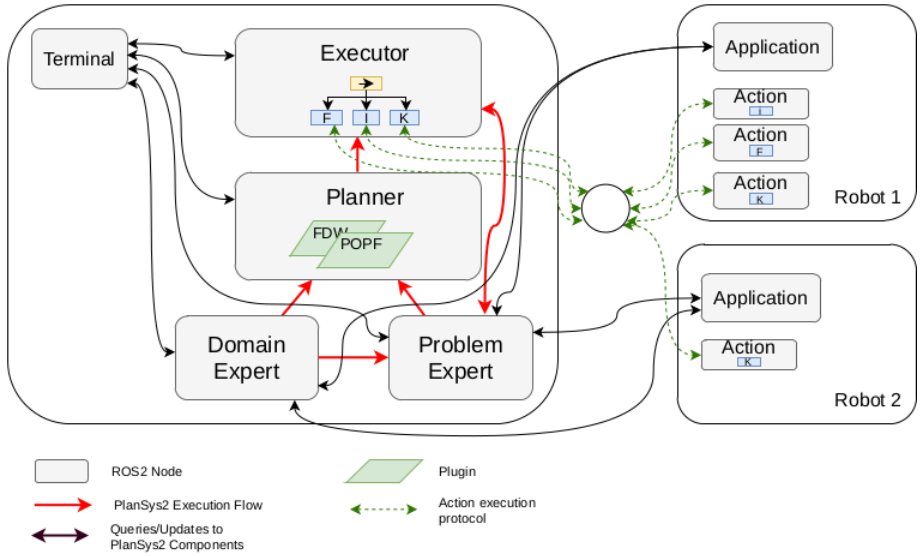
Similarly, we can configure the actions ask_charge and charge, with a sort of “fake” implementation

We need finally to add the nodes to the launch file!

For example:

```
move_cmd = Node(
        package='plansys_interface',
        executable='move_action_node',
        name='move_action_node',
        namespace=namespace,
        output='screen',
        parameters=[])

 ld.add_action(move_cmd)
```



Plansys2 Action Execution

The plan is now executed, and we can see from our node the progress of the different actions:

```
Action: (move robot1 corridor bedroom):10002 | Completion: 100% | Status:
SUCCEEDED
Action: (askcharge robot1 sitting_room corridor):0 | Completion: 100% | Status:
SUCCEEDED
Action: (charge robot1 corridor):5001 | Completion: 100% | Status: SUCCEEDED
Action: (move robot1 bedroom bathroom):15003 | Completion: 100% | Status:
SUCCEEDED
Everything done!!
```

When all actions are succeeded, the plan is over, and the node stops

Plansys2 Action Execution

What if we want to see the framework practically integrated with a robot?

We can think of modifying the “fake” move, to actually control a robot!

For example, we can think of using the info about the waypoint to reach to send a different velocity…

```
MoveAction()
  :
plansys2::ActionExecutorClient("mov
e", 250ms)
  {
    progress_ = 0.0;
    cmd_vel_ = this-
>create_publisher<geometry_msgs::
msg::Twist>("turtle1/cmd_vel", 10);
  }
```

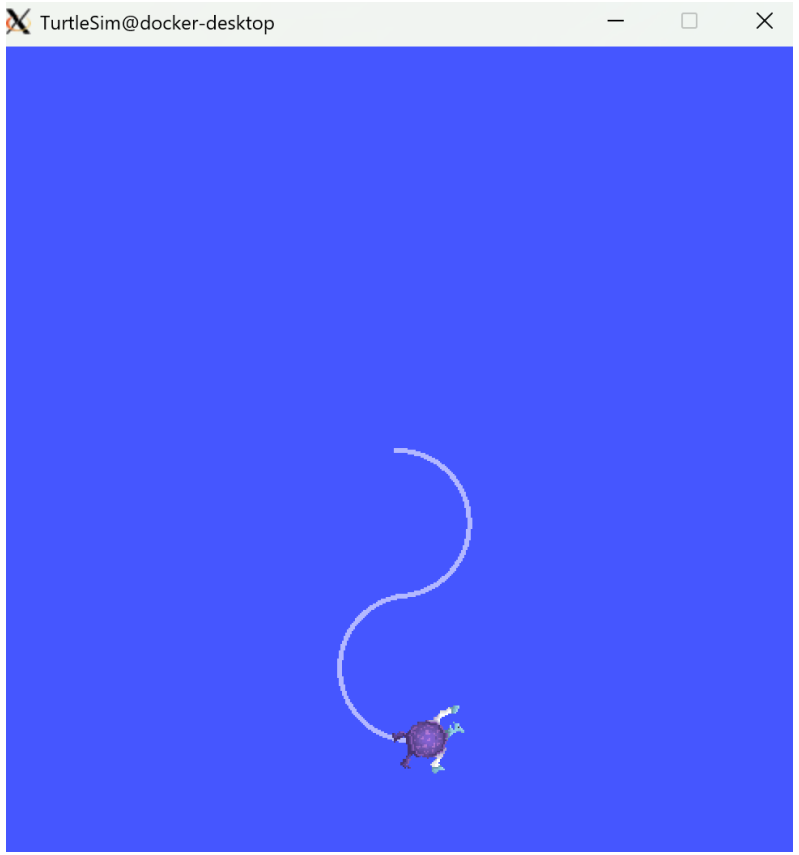
```
auto wp_to_navigate = args[2];
if (wp_to_navigate == "bathroom") {
¨}

if (wp_to_navigate == "bedroom") {
¨}
```

Plansys2 Action Execution

We can test this practical integration with a very simple simulator -> turtlesim

We also slightly change the launch file (distributed_actions_2.launch.py) so that now this “new” move action is used instead of the fake, simulated one.



ros2 launch plansys_interface distributed_actions_2.launch.py

ros2 run plansys_interface get_plan_and_execute

ros2 run turtlesim turtlesim_node

Plansys2 Action Execution

Finally, we could try to implement a more realistic approach, where the move action really performs navigation, and the waypoints (bedroom, bathroom, …) represent some x and y coordinates in the environment!

For this, we need to further modify our move action (move_action_node_3.cpp), by adding an action client to navigate_to_pose, i.e., the action server of the ros2 nav stack)

```
nav2_client_ = rclcpp_action::create_client<nav2_msgs::action::NavigateToPose>(
      nav2_node_, "navigate_to_pose"
    );
```

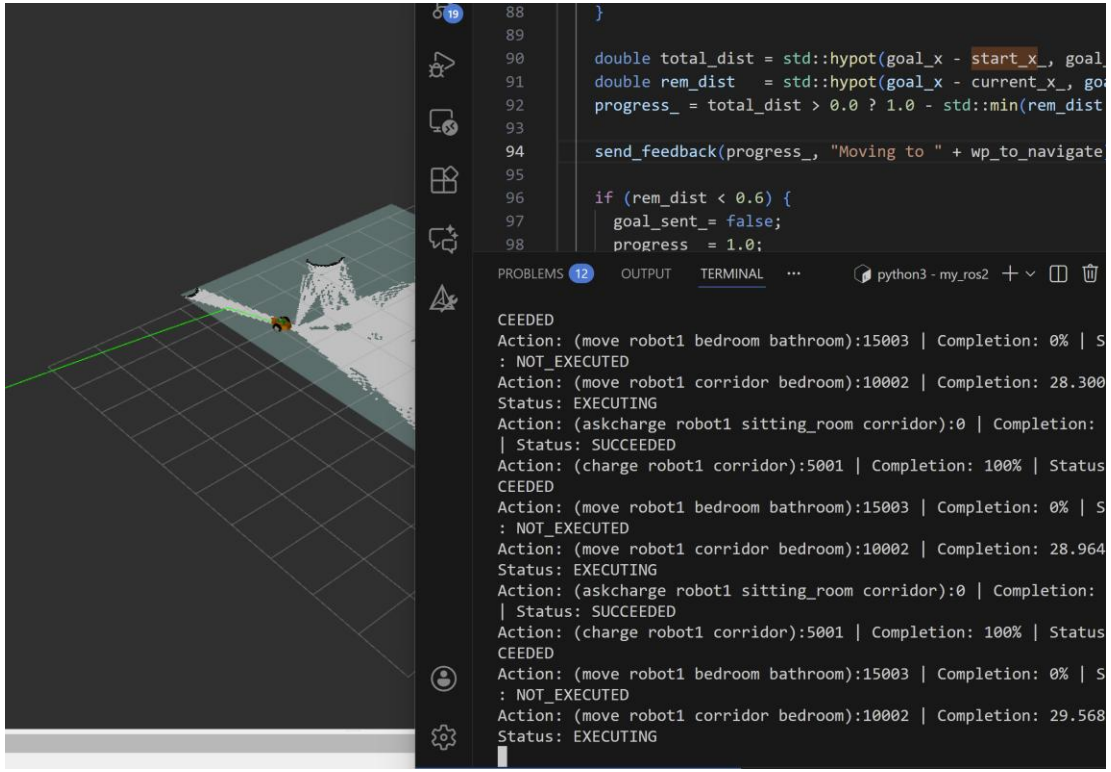
Based on the wp to be reached, the action node select the x,y goal and call the navigation server to reach the desired position.

Of course a similar work could be done to the other actions (ask_charge and charge)

Plansys2 Action Execution

To test the final integrated system, we need to run:

- ros2 launch plansys_interface distributed_actions_3.launch.py
- ros2 launch bme_gazebo_sensors spawn_robot.launch.py
- ros2 launch ros2_navigation mapping.launch.py
- ros2 launch ros2_navigation navigation.launch.py
- ros2 run plansys_interface get_plan_and_execute



Exercise

Based on the exercises shown in classes, create the plansys2-based package for a patrolling robot, that should periodically start from a given location, reach and inspect three waypoints in the enviroment, and go back to the starting position for a while.

Randomly, the *inspect* action in some waypoints can fail; when this happens, an alarm should be raised by the robot and it should stop patroling.

- *Draw a possible component diagram of the described system.*
- *Write the domain of the problem*
- *Write a launch file to test the proposed architecture*