Combined Task and Motion Planning API

Disclaimer: This document assumes knowledge of development in the Robot Operating System (ROS).

1. Description

This software library provides an interface between PDDL-based task planners and motion planners, following the framework laid out in [1]. Through use of the implemented interface layer, the task planner can operate in a geometry-independent abstract state space while handling geometric constraint discovery and requisite replanning via communication between the interface layer and the task and motion planners.

The interface layer has been implemented in python as a ROS package, which is set up to communicate with user-implemented task and motion planners via the client/server paradigm. More specifically, the task and motion planners must be implemented as ROS services that follow the specifications laid out in sections 3 and 4 for the task and motion planners respectively. The interface layer acts as client to both of these services. While the planners must comply with the I/O specifications and the task planner must be PDDL-based. no additional assumptions are made about their implementation. Figure 1 below diagrams the system framework.

Given that the pose generation and state update vary between domains, the user must also implement a domain specific pose generator class and state update function following the specifications laid out in section 2. These are to be written in python and passed to the interface layer upon its initialization.

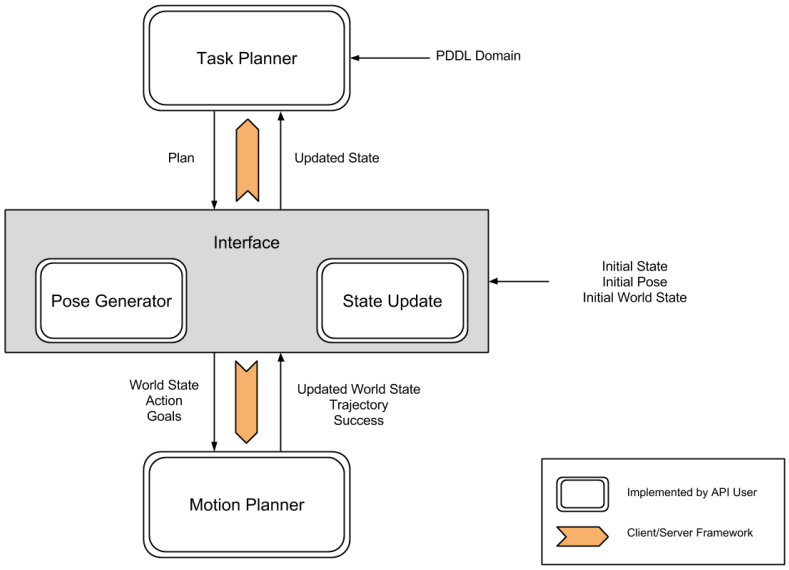

Figure 1: Combined Task and Motion Planning Framework

2. Interface

The `InterfaceLayer` class provides an `__init__` and a `run` public function.

```
class InterfaceLayer:
     def __init__(taskServerName, motionServerName, ...
                    poseGenerator, stateUpdate, ...
                         directory)
     def run(state, world, pose)
```

The `__init__` function takes as variables `taskServerName`, `motionServerName`, `poseGenerator`, `stateUpdate`, and `directory`; it returns an object of type `InterfaceLayer`. It is utilized in the following manner.

```
interfaceLayer = InterfaceLayer(taskServerName, ...
                                motionServerName, ...
                                poseGenerator, ...
                                stateUpdate, directory)
```

The `taskServerName` and `motionServerName` variables are strings specifying the names of their respective services. The `poseGenerator` variable is an instance of a user-defined pose generator class. The specification for this class is detailed in section 2.1. The `stateUpdate` variable is a user-defined state update function. The specification for this function is detailed in section 2.2. The directory variable is a string specifying the directory containing the initial state PDDL files.

The `run` function takes as variables `state`, `world`, and `pose`, which are the PDDL description of the initial state; the initial geometric state of the world, and the initial PDDL pose of the robot. This function communicates with the task and motion planners to find a viable plan and returns it as a tuple containing the high-level plan and the trajectory of the robot in that order. It is utilized in the following manner.

```
(hlplan,  traj) = interfaceLayer.run(state, world, pose)
```

The `state` variable must be given in the form similar to a PDDL file. The first line of the file must specify all of the objects, along with their types, separated by commas. The second line of the file must specify all of the initial predicates, separated by commas. The third line must specify the required goal predicates, separated by commas. All predicates are written without commas, just spaces between the different components of the predicates. There must not be spaces before the commas. The fourth line must be the initial pose. Everything must be written in all caps (except for types, where the case doesn't matter). A very small example is given below:

```
BLOCK0 - physob,S - location,ARM - gripper,INITPOSE - pose
AT BLOCK0 S,EMPTY ARM
IN BLOCK0 ARM
INITPOSE
```

The `world` variable is the description of the environment and the robot and is a custom ROS message of type `world_state`. The specification for this message is given in section 4. The `pose` variable specifies the initial pose of the robot, corresponding to the symbolic pose corresponding to the initial position of the robot in the PDDL file.

The `hlpan` variable takes the form of a list of tuples. Each tuple corresponds to an an action. For instance, consider the PICKUP action: the first item of the tuple will be the string 'PICKUP', followed by strings corresponding to the grasped object, the gripper, the initial pose, and the final pose. The `traj` variable is a list of trajectories as defined by the user in their motion planner.

2.1 Pose Generator

The pose generator class must return randomized, allowable poses. Specifically, the pose generator object must take in an action tuple, and based on the action, return an allowable pose with the next function. If poseGen is a pose generator object, it is called as poseGen.next(action), for action an action tuple. This function will return a valid pose only a fixed number of times, after which it returns 'None' to signify that the number of poses it can return is complete. This is necessary to signal the interface layer to backtrack.

When the interface layer backtracks, it needs to reset the pose generators to their original configurations where they can still return valid poses. This is done with the function poseGen.reset(action), to reset just the pose generator for a particular action, and poseGen.resetAll() to reset all of the pose generators at once.

Note that action is a full action tuple -- not a class of actions. So there should be a different poses returned and reset based on the full action, including the objects and parameters involved, not just for (for instance) all PICKUP actions.

2.2 State Update

The state update function must take in the current state of the world and update the state to reflect the currently reached state. This includes two parts. The first part is reading each of the high level plan actions since the last failed step and handling their effects by adding and removing predicates as necessary. This should echo exactly the effects of the action specified in the domain file for the PDDL.

Second, the state update function must add the appropriate obstructions based on the failCause. The failCause variable will be a list of the object id's (strings) for the objects obstructing the required action. Depending on the scenario, you need to create the appropriate predicate, generally a variant of OBSTRUCTS that relates to the current action being attempted.

Specifically, the inputs are:

state_input: the initial state before stateUpdate performs its function.
failCause: a list of objects which are obstructing a motion plan
failStep: the index into the high level plan where the obstruction occured
prev_fail_step: the index of the previous failure step
hlplan: the high level plan, in the normal data structure for high level plans
world: the current world, in its normal data structure format

3. Task Planner

The task planner must implement the following ROS service.

task_service.srv

```
task_domain domain
---
hl_plan plan
```

The input to the service is the `domain` variable, which complies with the custom ROS message type below. The variable `task_file` is the path to the file specifying the problem domain.

task_domain.msg

```
string task_file
```

The output of the service is the `plan` variable, which complies with the custom ROS message type below. The variable `error` is a bool specifying the success or failure of finding a viable task plan. The the case where `error` is false, the variable `plan` contains the plan found by the task planner. The plan string takes a very similar, pseudo-PDDL format to other data structures we include. Here, plan is a series of actions, with one action per line (ie, new line between the strings). Each action is written in all caps, with no commas, and with a space between each part. For instance, an action might look like: PICKUP OBJ1 GRIPPER POSE1 POSE2.

hl_plan.msg

```
bool error
string plan
```

4. Motion Planner

The motion planner must implement the following ROS service.

```
motion_service.srv
      motion_plan_parameters parameters
      ---
      motion_plan plan
```

The input to the service is the `parameters` variable, which complies with the custom ROS message type below. The variable `state` contains information about the geometric state of the environment and robot. The variable `action` specifies the action the motion planner must perform. The action is a tuple representing the PDDL action. For instance, consider the PICKUP action: the first item of the tuple will be the string 'PICKUP', followed by strings corresponding to the grasped object, the gripper, the initial pose, and the final pose. The variable `goals` is a list of goals generated by the user-implemented pose generator.

```
motion_plan_parameters.msg
      world_state state
      string action
      pose[] goals
```

The output of the service is the `plan` variable, which complies with the custom ROS message type below. The variable `state` contains information of about the updated geometric state of the world after the motion planner has carried out its actions. The variable `motion` contains information about the planned motion. The variable `success` is a bool specifying the success or failure of finding a motion plan.

```
motion_plan.msg
      world_state state
      motion_seq motion
      bool success
```

The contents of the `goals` variable in `motion_plan_parameters.msg` and the `motion` variable in `motion_plan.msg` are only ever accessed by the user defined modules (pose generator and motion planner; and motion planner respectively). Therefore, these message types can be written to contain the information most convenient to their implementation. These files can be found in the msg directory.

The `state` variable in `motion_plan_parameters.msg` and `motion_plan.msg` complies with the custom ROS message type below. The variable `world` contains information about the geometric state of the environment. The variable `robot` contains information about the geometric state of the robot.

```
world_state.msg
        world_obj world
        robot robot
```

The `world` variable in `world_state.msg` complies with the custom ROS message type below. It must contain a `movable_objects` list, which provides information about movable objects in the environment, but other variables useful to the motion planner can be included here. The file can be found in the msg directory.

```
world_obj.msg
        obj[] movable_objects
        ...
```

The `robot` variable in `world_state.msg` complies with the custom ROS message type below. It must contain an `id`, which gives the name of the robot, but other variables useful to the motion planner can be included here. The file can be found in the msg directory.

```
robot.msg
        string id
        ...
```

The `movable_objects` variable in `world_obj.msg` complies with the custom ROS message type below. It must contain an `id`, which gives the name of the object, but other variables useful to the motion planner can be included here. The file can be found in the msg directory.

5. Installation and Setup

Within the catkin workspace for an installation of ROS Indigo, clone the git repository https://github.com/bhomberg/6834-task-motion-planner in the proper location for a ROS package, e.g., /indigo_ws/src/.

Download FF version 2.3 from https://fai.cs.uni-saarland.de/hoffmann/ff/FF-v2.3.tgz (main page: https://fai.cs.uni-saarland.de/hoffmann/ff.html) and place the FF-v2.3 folder inside of the 6834-task-motion-planner directory.

Locations where directory of the 6834-task-motion-planner folder within the catkin workspace must be set:

benchmark.py: variable DIR_6834
small_tests.py: variable DIR_6834
planner_server.py: parameters passed in to TaskPlannerServer

interface_layer_organized.py: variable DIR

To run the tests, run the following commands in separate terminals:

```
roscore
rosrun task_motion_planner planner_server.py
rosrun task_motion_planner mockMotionPlanner.py
rosrun task_motion_planner small_tests.py
```

To run the benchmarks, just change the last line above to

```
rosrun task_motion_planner benchmark.py
```

6. References

[1] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, P. Abbeel. "Combined Task and Motion Planning Through an Extensible Planner-Independent Interface Layer". IEEE International Conference on Robotics and Automation (ICRA), 2014.