TDDE05: Lab 3: Task Execution

Cyrille Berger

March 3, 2020

The goal of this lab is to create a executor for Task-Specification-Trees. The TSTs are defined using the YAML format.

1 TST File format

TST are defined using the YAML file format, similar to JSON.

Bellow is an example, for a mission which repeat moving between two points:

```
{
  type: "repeat",
  child: {
    type: "sequence",
    children: [
      {
        type: "move_to",
        parameters: {
          use_motion_planner: true,
          position: [ 1, 20, 0 ],
          orientation: [ 0, 0, 0, 1 ]
        }
      },
      {
        type: "move_to",
        parameters: {
          use_motion_planner: true,
          position: [ -10, 0, 0 ],
          orientation: [ 0, 0, 1, 1 ]
        }
      }
    ]
  },
  parameters: {
    count: 2
  }
}
```

The *type* indicates the type of TST node, *child* and *children* are used by the container nodes and contains other node definition. *parameters* contains some parameter of the node.

2 TSTs node

2.1 Containers

Containers have either a *children* or *child* field which contains a list of sub nodes. There are three types of containers.

sequence Execute the children nodes in sequence.

concurrent Execute all the children nodes simultaneously. The parameters for that node are:

- *stop_on_first_finished*, optional, default to false, indicate that the concurrent node should stop execution when the first child node has finished execution. If it is set to false, the execution should wait until all the nodes are finished.

repeat Repeat the children nodes The parameters for that node are:

- *count*, optional, default to infinite, the number of times to repeat

2.2 Elementary actions

move_to : when executing this node, the robot should move to a specified location. The parameters for that node are:

- *use_motion_planner*, optional, default to false, indicate whether the action should use the motion planner to move to the location or not
- *position* vector of three elements specifying the target location
- *orientation* a quaternion with the targeted orientation of the robot

explore_spiral : explore the environment following a spiral trajectory, centered on the starting location of the robot, you can use the motion planner to avoid obstacles. The parameters for that node are:

- *radius* the radius of the spiral
- a and b representing the parameters of the spiral:

You should use the Archimedean spiral:

$$r = a + b\theta \tag{1}$$

record_topics : save topics in a ROS bag file[1]. The parameters for that node are:

- *topics* a list of topic to save
- *filename* the file where the topics are saved

ROS bag is a file format for saving messages published on the various topic. You can create a bag file with:

```
rosbag record -O <filename> <topic1> <topic2> ...
```

Then you can play it with (shutdown other ROS node before playing a bag file):

```
rosbag play <filename>
```

In Python you can use `subprocess.Popen` to start a process and the following snippet to stop the recoding:

```
def terminate_process_and_children(p):
  import psutil
  process = psutil.Process(p.pid)
  for sub_process in process.children(recursive=True):
    sub_process.send_signal(signal.SIGINT)
  p.wait()  # we wait for children to terminate
```

In C++, you can use `Boost.Process` library:

```
namespace bp = boost::process;

bp::child c("rosbag record ...");

...

c.terminate();
```

3 Implementation

For this lab, you have the choice to use C++ or Python. That choice is binding for subsequent labs. In case of doubt, Python is a better choice since the code for this lab is mostly going to be publishing on topics and calling service calls.

You should create a new package `air_lab3` (check *lab 0* or *lecture 2* if you are unsure how to create a package). You will need to depend on `rospy` or `roscpp` depending on your choice of programming language.

You should create a node which has a service call `execute_tst` which loads a TST description from a file and start executing it.

The details of the implementation of each nodes and of your executor is left to you. But a object-oriented design using the factory design pattern is a good idea.

[1] http://wiki.ros.org/rosbag

3.1 C++

For reading YAML using C++, you can use the *yaml-cpp* library: `https://github.com/jbeder/yaml-cpp`.

To use in CMakeLists.txt, you will need to explicitly link to the library with:

```
target_link_libraries(executable-name yaml-cpp)
```

```
#include <yaml-cpp/yaml.h>

YAML::Node test = YAML::LoadFile("tst.yaml");
```

To check how to access values, you can read a tutorial at `https://github.com/jbeder/yaml-cpp/wiki/Tutorial`.

3.2 Python

For reading YAML using Python, you can use the *pyyaml* library: `http://pyyaml.org/wiki/PyYAMLDocumentation`, example of use:

```
import yaml

with open('explore_spiral.yaml') as data_file:
  data = yaml.load(data_file)
```

4 Test cases

We provide some test cases:

```
roscd air_tsts/tsts
```

- `goto.yaml`: go to two positions in sequence
- `moveto_plan.yaml`: go to two positions in sequence using the motion planner
- `repeat_move_to.yaml`: repeat twice going to two positions
- `explore_spiral.yaml`: go to a start position and execute the spiral motion
- `explore_spiral_record.yaml`: go to a start position and execute the spiral motion and record some sensor data in a bag file