TDDE05: Lab 3: Task Execution

Cyrille Berger

March 9, 2021

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

1 TST

A Task-Specification-Tree (TST) is a hierachical tree structure which represents a complex mission. For the purpose of this lab, six types of nodes are used:

- *seq* allows to execute a sequence of nodes
- *conc* allows to execute a set of nodes concurrently, it has a parameter `stop-on-first` which controls whether the node should wait for all the children to finish or stop when the first node is done
- *drive-to* allows to send a ground robot to a specific position, it has the following parameters:
  - `p` the destination
  - `heading` the heading the robot must be facing at the end
  - `maximum-speed` the maximum speed that the robot need to drive
  - `use-motion-planner` whether the driving is done using the motion planner or not
- *echo* allows to publish a string on a topic, it takes the following parameters:
  - `topic` the name of the topic used for publishing
  - `text` the text published on the topic
- *explore* drives the robot in a spiral around the current location of the robot
  - `radius` the radius of the spiral
  - a and b representing the parameters of the spiral:

  You should use the Archimedean spiral:

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

  Do not try to use the motion planner while implementing this node.

- *record* records data in a bag file, it takes as parameters:
  - `filename` the name of the file where the data is saved
  - `topics` a space sperated list of topics to save

You can find the specification of the TST nodes in `air_tst/configs`[1], to access to the directory you can do:

```
roscd air_tst/configs
```

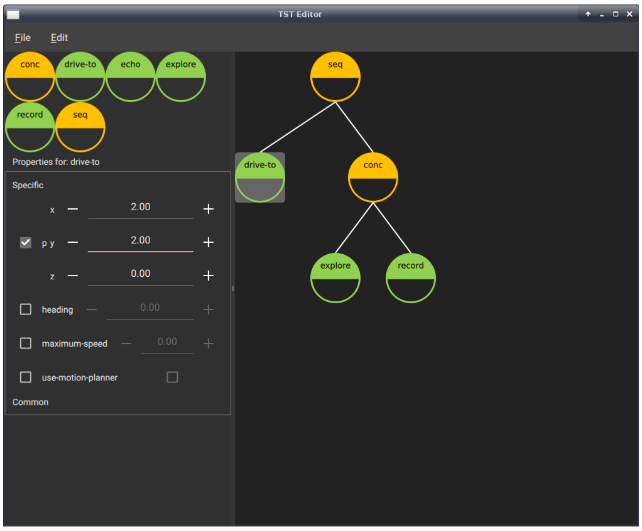
2 TST Editor

An editor is available in the `air_tst` module, and can be started with (and shown on figure 1):

```
rosrun air_tst tst_editor
```

You can open a specific file by giving it as an argument, for instance, the following will allow you to open the `explore_record.json` tst

```
roscd air_tsts/tsts
rosrun air_tst tst_editor explore_record.json
```



Properties for: drive-to

Specific

x — 2.00 +

☑ p y — 2.00 +

z — 0.00 +

□ heading — 0.00 +

□ maximum-speed — 0.00 +

□ use-motion-planner □

Common

Figure 1: TST Editor

3 TST File format

TST are defined using the JSON file format.

Bellow is an example, for an exploration mission, in which the robot first move to the location $(10, 2)$ and then explore in a spiral of radius 10 while recording some data:

[1] The directory on the lab computers is `/courses/TDDE05/software/catkin_ws/src/air_tst/configs`

```
{
    "children": [
        {
            "name": "drive-to",
            "params": {
                "p": {
                    "rostype": "Point",
                    "x": 10,
                    "y": 2,
                    "z": 0
                },
                "use-motion-planner": false
            }
        },
        {
            "children": [
                {
                    "name": "explore",
                    "params": {
                        "radius": 0
                    }
                },
                {
                    "name": "record",
                    "params": {
                        "filename": "explore.bag",
                        "topics": "/husky0/lidar"
                    }
                }
            ],
            "name": "conc"
        }
    ],
    "name": "seq"
}
```

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

4 TstML Library

The TstML library is a library that allows to parse the specification files, the TST files, provides some basic functionnality for manipulating TSTs and executing them. The generated API documentation is available in PDF on the course website. The document presents the C++ API, but the python API is similar. Examples of use of the TstML library are presented in the lab instruction, you only need to refer to the API if you have questions about the behavior of a specific function.

List of relevant classes:

- `TSTNode`: represents an instance of a node in a TST.
- `TSTNodeModel`: represents the model (or specification) of a type of TST Node.
- `TSTNodeModelsRegistry` registry of `TSTNodeModelsRegistry` used to define TSTs.
- `AbstractNodeExecutor`: base class to define the execution of a `TSTNode`.
- `ExecutionStatus`: holds the execution status
- `Executor`: handles the execution of a tree.
- `NodeExecutorRegistry`: contains an association between the `TSTNodeModel` and `AbstractNodeExecutor`. It is used during the execution of a TST to instantiate `AbstractNodeExecutor`.

5 Implementation

This lab can be completed using *Python* or *C++*. Pick the language you are the most comfortable with, but 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.

Follows the instruction corresponding to your choice of programming language.

5.1 air_lab3

Python You will need to create a package for `air_lab3`, in your `catkin_ws/src/air_labs` directory:

```
cd ~TDDE05/catkin_ws/src/air_labs
catkin_create_pkg air_lab3 air_tst rospy message_generation std_msgs
```

This will create a `air_lab3` package which depends on `air_tst` which is a package we provide with some basic definition and functionnalities, and on `rospy`.

C++ You will need to create a package for `air_lab3`, in your `catkin_ws/src/air_labs` directory:

```
cd ~TDDE05/catkin_ws/src/air_labs
catkin_create_pkg air_lab3 air_tst roscpp roslib message_generation std_msgs
```

This will create a `air_lab3` package which depends on `air_tst` which is a package we provide with some basic definition and functionnalities, and on `roscpp` and `roslib`.

5.2 tst_executor service

We will need to create a new service definition. This is very similar to the creation of message defintion (from lab 1). Instead, you should create a `ExecuteTst.srv` file in `air_lab3/srv`, with the following content:

```
string tst_file       # Name of the TST file to execute
---
bool success          # Whether the execution was successful
string error_message  # Error message if the execution was unsuccessful
```

Then in CMakeLists.txt, find the section with `add_service_files` and add your `ExecuteTst.srv` file. And uncomment the `generate_messages` section.

Then run `catkin build` and `start-tdde05`.

5.3 tst_executor node and service call

In your package, you should create a node called `tst_executor`. A node in ROS, in an executable.

In that node you need to define a service call `execute_tst` which loads a TST description from a file and start executing it.

Python Refer to lab 0 for how to create nodes in Python. You can look at lecture 2 for how to create a service server, as reminder:

```
def callback(req):
  return ServiceDefResponse()
service = rospy.Service('service_name', ServiceDef, callback)"
```

The service that you have created is available in the package `air_lab3.srv`.

C++ Refer to lab 2 for how to create nodes and service definition in C++. The service that you have created is available in the header `air_lab3/ExecuteTst.h` (the header is auto-generated).

In your `main` function, instead of calling `ros::spin()`, you should use the following:

```
ros::MultiThreadedSpinner spinner(4); // Use 4 threads
spinner.spin();
```

In your `CMakeLists.txt`, you should add:

```
set (CMAKE_CXX_STANDARD 17)
find_package(TstML REQUIRED)
```

And when you do `target_link_libraries`, you need to add `TstMLExecutor` to the list of libraries.

5.4 Initialise TstML and a load a TstML file

First we need to load the Tst definitions in a `TSTNodeModelsRegistry`.

Python You will need to create a `TstML.TSTNodeModelsRegistry` and use `loadDirectory` to load all the configuration file. It is best if this `tst_registry` is kept as a class member of your node.

```
import TstML
import rospkg

tst_registry = TstML.TSTNodeModelsRegistry()
tst_registry.loadDirectory(rospkg.RosPack().get_path("air_tst")
      + "/configs")
```

Then you can load a TST from a file:

```
tst_node = TstML.TSTNode.load(filename, tst_registry)
```

C++ You will need to create a `TstML::TSTNodeModelsRegistry` and use `loadDirectory` to load all the configuration file. It is best if this `tst_registry` is kept as a class member of your node.

```
#include <TstML/TSTNodeModelsRegistry.h>
#include <ros/package.h>

TstML::TSTNodeModelsRegistry* tst_registry
    = new TstML::TSTNodeModelsRegistry();
tst_registry->loadDirectory(
    QString::fromStdString(
          ros::package::getPath("air_tst") + "/configs"));
```

Then you can load a TST from a file:

```
TstML::TSTNode* tst_node = TstML::TSTNode::load(
    QUrl::fromLocalFile(QString::fromStdString(filename)),
        tst_registry);
```

5.5 Initialise TstML::Executor and start execution

python First you need to create a `NodeExecutorRegistry`, and use the `registerNodeExecutor` function to associate `TSTNodeModel` with `AbstractNodeExecutor`. We provide you with a default executor for `sequence` and `concurrent`.

```python
import TstML.Executor

# Create a registry with node executors
tst_executor_registry = TstML.Executor.NodeExecutorRegistry()

# Setup the executors for sequence and concurrent
tst_executor_registry.registerNodeExecutor(
      tst_registry.model("seq"),
      TstML.Executor.DefaultNodeExecutor.Sequence)
tst_executor_registry.registerNodeExecutor(
      tst_registry.model("conc"),
      TstML.Executor.DefaultNodeExecutor.Concurrent)
```

Now you can do the following to execute nodes

```python
# Create an executor using the executors defined
# in tst_executor_registry
tst_executor = TstML.Executor.Executor(tst_node,
      tst_executor_registry)

# Start execution
tst_executor.start()

# Block until the execution has finished
status = tst_executor.waitForFinished()

# Display the result of execution
if status.type() == TstML.Executor.ExecutionStatus.Type.Finished:
  print("Execution successful")
else:
  print("Execution failed: {}".format(status.message()))
```

Instead of displaying the results, you should set the result in the response of your service call.

C++ First you need to create a `NodeExecutorRegistry`, and use the `registerNodeExecutor` function to associate `TSTNodeModel` with `AbstractNodeExecutor`. We provide you with a default executor for `sequence` and `concurrent`.

```cpp
#include <TstML/Executor/NodeExecutorRegistry.h>
#include <TstML/Executor/DefaultNodeExecutor/Sequence.h>
#include <TstML/Executor/DefaultNodeExecutor/Concurrent.h>

// Create a registry with node executors
TstML::Executor::NodeExecutorRegistry* tst_executor_registry
    = new TstML::Executor::NodeExecutorRegistry();
```

```

// Setup the executors for sequence and concurrent
tst_executor_registry->registerNodeExecutor
    <TstML::Executor::DefaultNodeExecutor::Sequence>
        (tst_registry->model("seq"));
tst_executor_registry->registerNodeExecutor
    <TstML::Executor::DefaultNodeExecutor::Concurrent>
        (tst_registry->model("conc"));
```

Now you can do the following to execute nodes

```
// Create an executor using the executors defined
// in tst_executor_registry
TstML::Executor::Executor* tst_executor
    = new TstML::Executor::Executor(tst_node, tst_executor_registry)

// Start execution
tst_executor->start();

// Block until the execution has finished
TstML::Executor::ExecutionStatus status = tst_executor->waitForFinished();

// Display the result of execution
if(status == TstML::Executor::ExecutionStatus::Finished())
{
  ROS_INFO("Execution successful");
}
else
{
  std::string message = status.message().toStdString();
  ROS_INFO("Execution successful '%s'", message.c_str());
}

delete tst_executor;
```

Instead of displaying the results, you should set the result in the response of your service call.

5.6 Implement an AbstractNodeExecutor

The main goal of the lab is to implements `AbstractNodeExecutor` for the terminal nodes. In this section, I will give you the implementation of the `echo` executor:

python The following show an example of implementation for `echo` (see `echo_executor.py` on the website):

```python
import std_msgs.msg
import time

class EchoExecutor(TstML.Executor.AbstractNodeExecutor):
  def __init__(self, node, context):
    super(TstML.Executor.AbstractNodeExecutor, self).__init__(node,
          context)
    # Create publisher
    self.pub = rospy.Publisher(node.getParameter(
          TstML.TSTNode.ParameterType.Specific,
          "topic"), std_msgs.msg.String, queue_size=1)
    # Make sure that ROS publisher/subscriber are established
    time.sleep(2.0)
    # Counter
    self.count = 0
    self.paused = False

  def start(self):
    msg = std_msgs.msg.String()
    msg.data = self.node().getParameter(
          TstML.TSTNode.ParameterType.Specific, "text")

    finite_loop = self.node().hasParameter(
          TstML.TSTNode.ParameterType.Specific, "count")
    loop_count = self.node().getParameter(
          TstML.TSTNode.ParameterType.Specific, "count")

    # Called at a regular interval to publish on the topic
    def callback(event):
      if not self.paused:
        self.pub.publish(msg)
        self.count += 1
        if finite_loop and self.count >= loop_count:
          self.executionFinished(
              TstML.Executor.ExecutionStatus.Finished())
          self.timer.shutdown()

    # Create a timer
    self.timer = rospy.Timer(rospy.Duration(self.node().getParameter(
          TstML.TSTNode.ParameterType.Specific, "interval")),
          callback)

    return TstML.Executor.ExecutionStatus.Started()
  def pause(self):
    self.paused = True
    return TstML.Executor.ExecutionStatus.Paused()
  def resume(self):
    self.paused = False
    return TstML.Executor.ExecutionStatus.Running()
  def stop(self):
    self.timer.shutdown()
    return TstML.Executor.ExecutionStatus.Finished()
  def abort(self):
    self.timer.shutdown()
    return TstML.Executor.ExecutionStatus.Aborted()

# Associate the EchoExecutor to the "echo" model
tst_executor_registry.registerNodeExecutor(
      tst_registry.model("echo"), EchoExecutor)
```

C++ The following show an example of implementation for `echo` (see `echo_executor.cpp` on the website):

```
#include <std_msgs/String.h>

#include <QTimer>

#include <TstML/Executor/AbstractNodeExecutor.h>
#include <TstML/Executor/AbstractNodeExecutor.h>
#include <TstML/TSTNode.h>

class EchoExecutor : public TstML::Executor::AbstractNodeExecutor
{
public:
  EchoExecutor(const TSTNode* _node,
      AbstractExecutionContext* _context) :
      TstML::Executor::AbstractNodeExecutor(_node, _context)
  {
    // Create publishjer
    m_pub = m_nodeHandle.advertise<std_msgs::String>(
        node()->getParameter(TstML::TSTNode::ParameterType::Specific,
            "topic").toString().toStdString(), 1);
    // Make sure that ROS publisher/subscriber are established
    QTimer::sleep(2);
    m_count = 0;
    m_paused = false;
    // Get paremeters
    m_finite_loop = node()->hasParameter(
          TstML::TSTNode::ParameterType::Specific, "count");
    m_loop_count = node()->getParameter(
          TstML::TSTNode::ParameterType::Specific, "count").toInt();
  }
  // Callback for the timer used to publish the string
  void callback(const ros::TimerEvent&)
  {
    if(m_paused) return;
    std_msgs::String msg;
    msg.data = node()->getParameter(
          TstML::TSTNode::ParameterType::Specific, "text")
          .toString().toStdString();
    m_pub.publish(msg);
    ++m_count;
    if(m_finite_loop and m_count >= m_loop_count)
    {
      executionFinished(TstML::Executor::ExecutionStatus::Finished());
      m_timer.stop();
    }
  }
  TstML::Executor::ExecutionStatus start() override
  {
    m_timer = m_nodeHandle.createTimer(
        ros::Duration(node()->getParameter(
          TstML::TSTNode::ParameterType::Specific, "interval").toInt()),
          &EchoExecutor::callback, this);
    return TstML::Executor::ExecutionStatus::Started();
  }
  TstML::Executor::ExecutionStatus pause() override
  {
    m_paused = true;
    return TstML::Executor::ExecutionStatus::Paused();
  }
  TstML::Executor::ExecutionStatus resume() override
  {
    m_paused = false;
    return TstML::Executor::ExecutionStatus::Running();
  }
  TstML::Executor::ExecutionStatus stop() override
  {
    m_timer.stop();
    return TstML::Executor::ExecutionStatus::Finished();
  }
  TstML::Executor::ExecutionStatus abort() override
  {
    m_timer.stop();
    return TstML::Executor::ExecutionStatus::Aborted();
  }
private:
  ros::NodeHandle m_nodeHandle;
  ros::Publisher m_pub;
  ros::Timer m_timer;
  int m_count;
  int m_paused;
  bool m_finite_loop;
  int m_loop_count;
}
// Associate the EchoExecutor to the "echo" model
tst_executor_registry->registerNodeExecutor(
      tst_registry.model("echo"), EchoExecutor)
```

5.7 Test execution

You should now start your `tst_executor` node, and we can use a service call to execute our first TST:

```
rosservice call /husky0/execute_tst \\
  "tst_file: '`rospack find air_tsts`/tsts/echo.json'"
```

You can use the following to check was is echoed:

```
rostopic echo /husky0/test
```

5.8 Implement abort/stop/pause/resume

You can use `std_srvs/Empty` to define a service call for `abort`, `stop`, `pause` and `resume`. Those service call need to be defined in your `tst_executor` node, along the `execute_tst` service call. You can call the function `abort`, `stop`, `pause` and `resume` of the `TstML.Executor.Executor/TstML::Executor::Executor` class.

6 Test cases

We provide some test cases:

```
roscd air_tsts/tsts
```

- `echo.json`: publish on a topic
- `drive_to.json`: go to three positions in sequence

- drive_to_repeat.json: go to three positions in sequence and repeat once
- explore.json: go to a start position and execute the spiral motion
- explore_record.yaml: go to a start position and execute the spiral motion and record some sensor data in a bag file

7 Implementation of Executors

You need to implement the drive_to, explore and record node. It involves to mostly publish and listen.

Drive to Need to support both using the motion planner or not. For setting the maximum speed, use $maximum-speed$ for the linear speed and $3.0 * maximum-speed$ for the angular one, and if none is set, default to $maximum-speed = 0.5$. Heading is equivalent to Yaw, for C++ you can look at lab 2 for conversion between Yaw and Quaternion. In Python:

```python
import tf
quat = tf.transformations.quaternion_from_euler(0, 0, yaw)
# quat is an array with the coordinates as [x,y,z,w]
```

You might need to modify your state machine to send an event when the position has been reached and when the waypoint controller has finished:

- add two output events, called position_reached and waypoints_finished
- connect the finished event of the idle of the reach point controller to position_reached (like you did to the pop state)
- connect the empty event of pop to waypoints_finished

Look at Figure 2 for an example of how it could look like. Then waypoints_finished and position_reached are available as topics with type std_msgs/Empty.

After creating your publisher/subscriber, you should sleep, so that all the connections are properly established by ROS.

In python, you can sleep with:

```python
import time
time.sleep(2.0)
```

In C++, you can:

```cpp
#include <QThread>
QThread::sleep(2);
```

Explore sample the Archimedean spiral. You can increment θ by $\pi/4$ to generate the waypoints until r is superior to the $radius$ given as argument.

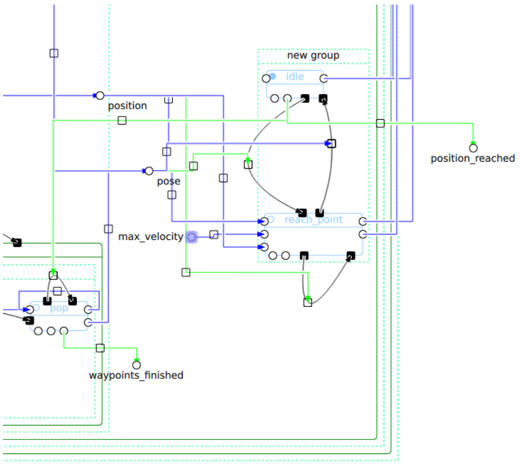


Figure 2: Event connections

record save topics in a ROS bag file[2]. The parameters for that node are:

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

For this node, it will be important to implement the `stop` function correctly.

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):
  """ Take a process as argument, and kill all the children
  recursively.
  """
  import psutil
  import signal
  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
```

Example of use:

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

```python
import subprocess
p = subprocess.Popen(["rosbag", "record", "/rosout"])
...
terminate_process_and_children(p)
```

In C++ , you can use `QProcess`:

```cpp
#include <QProcess>

QProcess p;
p.start("rosbag", {"record", "..."});
...
p.terminate();
```

8 Launch file

Update your launch file to include `ls_to_occ`, `occ_to_display`, `motion_planner` and `move_to_point.py`!

8.1 Demonstration

- Show your `launch` file
- Show the execution of `drive_to_repeat.json` and `explore_record.yaml`