APPENDIX A

APPENDIX: USER'S GUIDE

1. System Requirements:

   LTLstack can be used under the environment of Ubuntu 14.04 with ROS Indigo installed or Ubuntu 16.04 with ROS Kinetic installed.

2. Versions:

   LTLstack-Indigo can be installed under Ubuntu 14.04.

   `https://github.com/VerifiableRobotics/LTL_stack/tree/master`

   LTLstack-Kinetic can be installed under Ubuntu 16.04.

   `https://github.com/VerifiableRobotics/LTL_stack/tree/ros-kinetic`

3. Dependencies:

   The following dependencies are required: 1.slugs 2.Tkinter

   How to install slugs:

   (a) Install the specific version of slugs: `git clone -b LTL_stack https://git@github.com/wongkaiweng/slugs.git`

   (b) Install the dependence: `sudo apt-get install libboost-all-dev`

   (c) Compile slugs by running `make` in the *slugs/src* directory

   (d) Make sure slugs can be found anywhere by adding this line `export PATH=:$PATH:</path/to/slugs-src-folder>` to your ~ */.bashrc* file.

   How to install Tkinter: `sudo apt-get install python-tk`

4. Try an example:

   (a) Download the repo into your workspace *src* folder: `git clone https://github.com/VerifiableRobotics/LTL_stack.git`. After the master branch is downloaded, pull the `ros-kinetic` branch to have the latest version.

   (b) Find the directory to your workspace and run: `catkin_make`

   (c) Check if all the LTLstack nodes are executable. If not, run `find <dir to LTL_stack> -type f -exec chmod 755 {} \;`. Replace the `<dir to LTL_stack>` to your LTLstack folder.

   (d) Try to run the Rotating Turtle example! And have fun!

      run `roslaunch controller_executor tutorial_all.launch`

5. How to write a proposition node:

   There are two main kinds of ROS node, the publisher node and the subscriber node. In LTLstack, a publisher node, aka. system (actuator) node, is a node sending messages to the robotic platform, while a subscriber node, aka. environment (sensor) node, is a node receiving messages from the environment sensors.

   (a) Publisher (system) node:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

According to the publisher node example above, let's read in detail to learn how the node is made.

```
#!/usr/bin/env python
```

Every Python ROS node should have this declaration at the top. The first line makes sure your script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

You need to import `rospy` if you are writing a ROS node. The *std_msgs.msg* contains the basic message type we can use, and in this example, the `String` is used.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

This section of the code defines the publisher, including which topic to publish, the message type in the topic, and how many pieces of message can be in the topic at the same time. The second line defines the node's name. When your node's name is taken by previous ROS nodes, this section `anonymous=True` can help you to unique your ROS node by adding random number to the end of the node's name.

```
    rate = rospy.Rate(10) # 10hz
```

This line helps you to define in which frequency you want your message to be published.

```
 while not rospy.is_shutdown():
     hello_str = "hello world %s" % rospy.get_time()
     rospy.loginfo(hello_str)
     pub.publish(hello_str)
     rate.sleep()
```

This loop keeps the node to continuously publish message until the *roscore* is shut down. `rate.sleep()` has very similar function as `time.sleep()` in python.

```
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The main block triggers the function, `talker()`, and try to catch the exception.

(b) Subscriber (environment) node:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String


def callback(data):
        rospy.loginfo(rospy.get_caller_id() + "I heard %s"
        ..., data.data)

    def listener():
       rospy.init_node('listener', anonymous=True)
       rospy.Subscriber("chatter", String, callback)
       rospy.spin()

  if __name__ == '__main__':
       listener()
```

The subscriber node is very similar to the publisher node. The biggest difference is the additional `callback` function, which is not shown in the publisher node.

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
rospy.spin()
```

This section of the node initializes the node and "give" the node its role, the subscriber. The node subscribes the message from the topic `chatter` in the type of `String`. The message is coming from the `callback` which is defined in the following section.

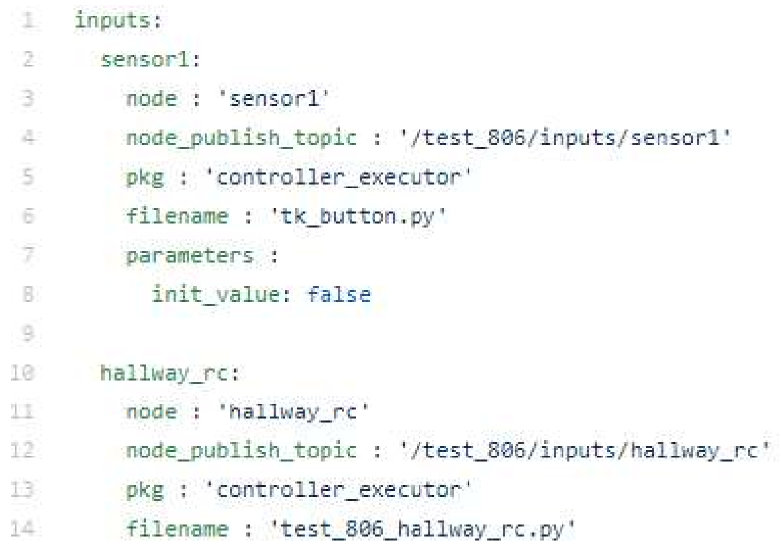
```
def callback(data):
     rospy.loginfo(rospy.get_caller_id() + "I heard %s"
     ..., data.data)
```

This is a python function, inside this function is how the message is generated. We can also make another subscriber node here to subscribe to another topic. In this way, we can make a connection among three or more nodes.

6. Files needed to make a LTLstack package:

   (a) A specification in *.slugsin* format. In this thesis, the specification can be generated by LTLMoP.

   (b) A YAML file that maps the ROS proposition nodes with the inputs/outputs of the specification. Below is a YAML file from example 1 in the thesis.



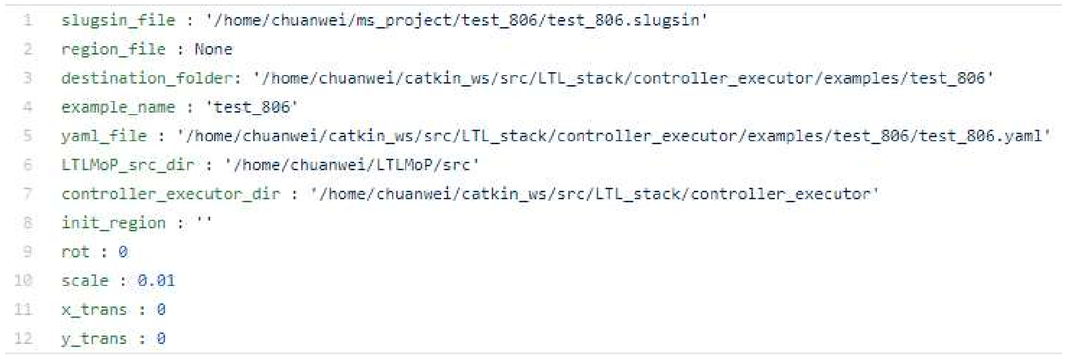
```
inputs:
  sensor1:
    node : 'sensor1'
    node_publish_topic : '/test_806/inputs/sensor1'
    pkg : 'controller_executor'
    filename : 'tk_button.py'
    parameters :
      init_value: false

  hallway_rc:
    node : 'hallway_rc'
    node_publish_topic : '/test_806/inputs/hallway_rc'
    pkg : 'controller_executor'
    filename : 'test_806_hallway_rc.py'
```

Figure A.1: An example of the YAML file in LTLstack

   (c) Create a *SetupLaunch* YAML File. This is the file to specify where the *.slugsin* file is, where all the ROS nodes are and where the previous YAML file is. Below is a YAML file from example 1 in the thesis.

7. How to create a new example:

   (a) Create environment (sensor)/system (actuator) ROS nodes following the instruction of Appendix A.5



```
slugsin_file : '/home/chuanwei/ms_project/test_806/test_806.slugsin'
region_file : None
destination_folder: '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor/examples/test_806'
example_name : 'test_806'
yaml_file : '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor/examples/test_806/test_806.yaml'
LTLMoP_src_dir : '/home/chuanwei/LTLMoP/src'
controller_executor_dir : '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor'
init_region : ''
rot : 0
scale : 0.01
x_trans : 0
y_trans : 0
```

Figure A.2: An example of the setup YAML file in LTLstack

(b) Create the Proposition YAML File following the instruction of Appendix A.6

(c) Create the setup YAML File following the instruction of Appendix A.6

(d) With the two YAML files, you can automatically generate all the launch files with the following command:

```
rosrun controller_executor setup_launch_file_with_yaml.py
[setup_launch_yaml_file_dir/setup_launch_yaml_file_name]
```

8. How to run a new example:

After all the previous steps, running an example is straight forward. Just run: `roslaunch controller_executor <your example name>_all.launch`. Replace the `<your example name>` with your example name.

APPENDIX B

APPENDIX: DEBUG A LTLSTACK PACKAGE

To debug a LTLstack package, we can use the following decision tree, fig B.1 as guide to find and fix the bugs.

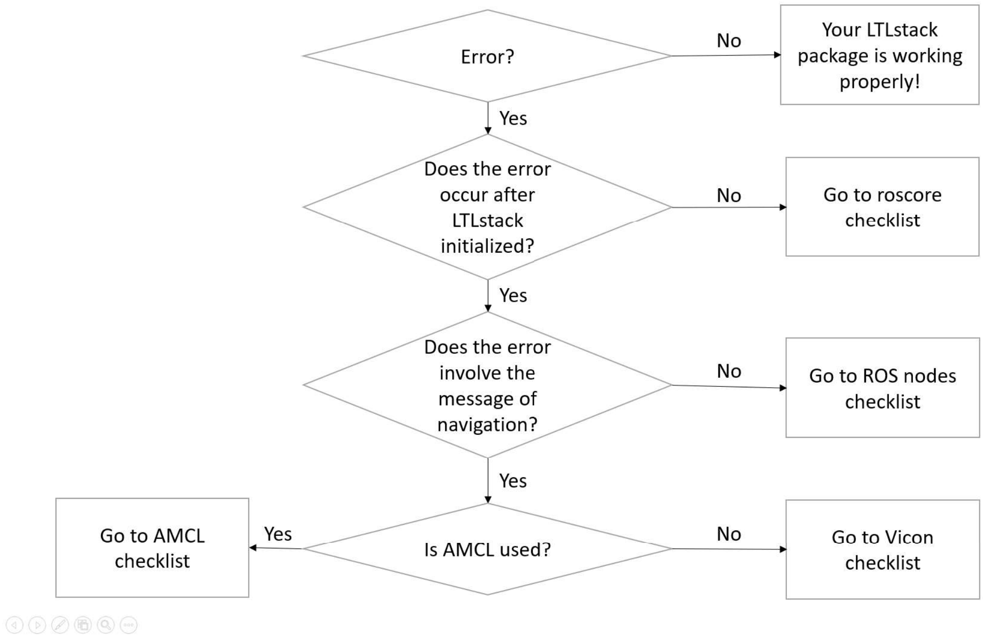

Error?
No
Your LTLstack package is working properly!
Yes
Does the error occur after LTLstack initialized?
No
Go to roscore checklist
Yes
Does the error involve the message of navigation?
No
Go to ROS nodes checklist
Yes
Go to AMCL checklist
Yes
Is AMCL used?
No
Go to Vicon checklist

Figure B.1: The decision tree for the debug process for LTLstack package

`roscore` Checklist:

1. Is another computer in the same wifi environment running the `roscore`?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the `ROS_MASTER_URI` setting right in `/.bashrc`?

   The `/.bashrc` interprets your typed input in the Terminal program and runs commands based on your input. To trigger the right `ROS_MASTER_URI` while `roscore` is called, you should insert `export ROS_MASTER_URI=http://<the IP address where the roscore should be running on>:11311/` in `/.bashrc`. Change `<the IP address where the roscore should be running on>` to your desired IP address. Don't forget to restart all the terminals after the `/.bashrc` is changed, since the changes wouldn't affect the existing terminals.

ROS node checklist:

Go to `<your ROS nodes dir>`, run `python <the related ROS node>.py` to debug the related ROS node based on the information provided via python. Change `<your ROS nodes dir>` to where the ROS nodes are stored. Change `<the related ROS node>` to the ROS node you wanted to analyze.

AMCL checklist:

1. Is another computer in the same wifi environment running the AMCL package?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the AMCL package turned on?

   If not, run the AMCL package to trigger the package. The AMCL package vary depends on different robotic platform is used. For example, if Jackal is used, you would run `roslaunch jackal_navigation amcl_demo.launch map_file:=<path to your map>.yaml`. Change `<path to your map>` to your map.

3. Are the related AMCL topics established properly?

   To check this, run `rostopic list` to see if the desired AMCL topics are listed.

4. Are the related AMCL topics publishing the proper message?

   To check this, run `rostopic echo <the topic you want to analyze>` to see if the vicon topic was publishing the proper message. Change `<the topic you want to analyze>` to your ROS AMCL topic. Here are some commonly used AMCL topics: `amcl_pose`, robot's estimated pose in the map; `particlecloud`, the set of pose estimates being maintained by the filter; `tf`, publishes the transform from odometry to map.

Vicon checklist:

1. Is another computer in the same wifi environment running the `vicon_bridge` package?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the vicon package turned on?

   If not, run `roslaunch vicon_bridge vicon.launch` to trigger the package.

3. Are the desired vicon objects checked?

   If not, check the related vicon objects in the vicon system, then click the "track" button in the vicon system.

4. Are the related vicon topics established properly?

   To check this, run `rostopic list` to see if the desired vicon topics are listed.

5. Are the related vicon topics publishing the proper message?

   To check this, run `rostopic echo <the topic you want to analyze>` to see if the vicon topic is publishing the proper message. Change `<the topic you want to analyze>` to your ROS vicon topic.

6. Is the transform package properly used?

   To check this, run `rosrun tf view_frames`. This would provide you the information about how relationship between coordinate frames is made in the ROS system. Usually, if the above 5 check points don't give you any unexpected result, there are problems hidden in the transform relationship related to vicon.