Simulation-based Verification of Functions for Autonomous Drones

Hamza Ghezali[2], Siddhartha Gupta[2], Umut Durak[1]

[1]Institute of Flight Systems, German Aerospace Center (DLR), Lilienthalplatz 7, 38108 Braunschweig, Germany
[2]Institute of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany

Abstract. The world is witnessing considerable interest and a rapid shift towards the autonomous aerial domain. Aerial robotics brings a lot of solutions to some existing problems and has benefits in many fields, such as transportation of people and goods, navigation, military defence, games, agriculture and so on. Small aerial robots, also known as drones, have critical applications due to their flexibility and suitable small size. Developing these small electronic devices is an arduous task involving careful planning and testing. In addition, Their development and delivery to end users should not only be finished in a shorter amount of time and also with high liability. One of the main concepts to address this issue is using a new testing technique called Behavior Driven Development (BDD). This work uses BDD in component-based software engineering for aerial robotics as an agile technique. This testing approach offers an exciting method to write different possible scenarios in simple, easy-to-read, and understandable feature files. An example of applying this method for drone application development with Robot Operating System and Gazebo is shown in this paper.

Introduction

We live in a time where "flying robots" are no longer only in science fiction movies. In the beginning, aerial robots were mainly considered for military surveillance, but currently, they have a vital role in targeting and special missions. Over time their involvement in the civilian domain has drastically increased. Some examples are aerial photography, remote sensing, agricultural and wildlife surveys, disaster response, and delivering products. Aerial robots are complex Cyber-Physical Systems (CPS) with various kinds and many sensors, actuators, controllers, and communication systems. Robot Operating Systems (ROS) is a framework for large-scale robotic applications which offers a peer-to-peer communication infrastructure. It uses a central broker for service registration and discovery [1]. It is becoming a famous architecture for developing software components and aerial robotics. Some of the recent studies on the utilisation of ROS for aerial robotics include ([2]; [3]; [4]; [5]). There are testing strategies developed in the industry for ROS-based robotics software already ([6]; [7]; [8]). Most of them follow the Test Driven Development (TDD) approach, which has its roots in agile methods [9]. Modern software engineering practices tend to remove disconnects among its activities by employing continuous practices to achieve agile processes. After Test-Driven Development (TDD) bridged the gap between implementation and testing, Behavior Driven Development (BDD) aims at the gap between the end-user and the implementation. It establishes a practice based on behavior specifications from the end-user's perspective. It builds upon TDD and promotes a ubiquitous semi-formal language for the specification of behavior that is accessible to all the stakeholders of the system. BDD aims to come up with an executable and a human-readable specification of the system, which can be used for automated acceptance testing. There is a growing interest in the aerial robotics domain towards agile software engineering methods. However, being complex Cyber-Physical Systems, system-level testing of aerial robots requires closed-loop simulation-based approaches [10]. Gazebo is an open-source robot simulator for developing algorithms, designing robots, performing tests, or training AI systems using realistic scenarios ([11]; [12]). Gazebo-ROS integration has also been developed in the last decade [13]. It has then been used also for simulations of aerial robotics ([14]; [15]; [16]; [2]). This paper will present an approach that integrates BDD with Gazebo to enable agile simulation-based verification of ROS-based applications for aerial robotics.

1 Background

1.1 Behavior Driven Development

BDD is defined as "implementing an application by describing its behavior from the perspective of its stakeholders" [17]. It promotes a ubiquitous semi-formal language for the specification of behaviors that is accessible to all the stakeholders of the system. It is structured around features, which can be defined as the system's capabilities that benefit its users. A feature is usually described in BDD by a title, a brief narrative, and a number of scenarios that serve as acceptance criteria. Scenarios are concrete examples to illustrate the desired behaviors of the system. When the concrete examples are executable, they turn the criteria into an acceptance test. BDD calls this automated acceptance testing. The ubiquitous language idea is based on Evans [18], who stresses that the linguistic divide or the language fracture between the domain expert and the technical team leads to only vaguely described and vaguely understood requirements. Gherkin is the common language for writing features [19]. While it is not a Turing Complete language, it has a grammar enforced by a parser. Each Gherkin feature may contain an arbitrary number of scenarios, which serve as acceptance criteria for the feature. A scenario typically consists of multiple steps describing the actions needed to stimulate the scenario and check the outcome. The steps of a scenario are distinguished into three different classes.

• Given prefaces, a step describes some initial state of the application or the world surrounding the application.

• When introducing a mutation of state on the application or the world. Often this can also be described as the occurrence of an event.

• Then is used to describe an expected result after the arrival of an event in a prior When step.

1.2 Behave Framework

Behave [20] is a framework written in python to perform BDD and allows the development of test cases using the Gherkin language. Behave is opiated on source file organisation where a test case is placed inside the directory feature. Multiple features can be put into different files having the extension *.feature*. A feature may have multiple steps, and those step implementations reside in a nested directory called steps. A background consists of a series of steps like scenarios. It allows you to add some context to the scenarios of a feature. A background is executed before each scenario of this feature but after any before hooks. The background description is for the benefit of humans reading the feature. Again, the background name should be a descriptive yet brief title, illustrating the background operation being performed or the requirement being met. A background section may exist only once within a feature file. In addition, a background must be defined before any scenario or scenario outline. Scenarios describe the discrete behaviours being tested. They are given a title, like backgrounds and the scenario description, for the benefit of humans reading the feature text. Scenario outlines may be used when you have expected conditions and outcomes to go along with your scenario steps. An outline utilises placeholders in the step definition, replaced by values from a table. You may have several example tables in each scenario outline. Steps take a line each and begin with a keyword - one of "Given", "When", "Then", "And", or "But". In a formal sense, the keywords are all Title Case, though some languages allow all-lowercase keywords where that makes sense.

1.3 Robot Operating System (ROS)

ROS is an open-source meta-operating system to ease the process of building a robot. ROS shares similarities with traditional operating systems by having process management and scheduling features. "It provides a structured communications layer above the host operating systems of a heterogeneous compute cluster" [21]. ROS is currently one of the most popular platforms for the collaboration of robotics solutions. The key concepts in ROS are nodes, ROS master, topics, messages and services.

Nodes: They are the smallest functional unit inside ROS architecture. They execute a particular functionality and help to achieve modularity. They can be thought of as processes. Just as a normal operating system manages processes and their communication, ROS manages nodes and communication between them.

Messages: Nodes communicate through messages. ROS has over 200 predefined messages that may contain simple values and more complex data like text, images, etc. Users also can create their messages.

ROS master: ROS supports a peer-to-peer network of ROS nodes which communicate through messages. There is an individual node called ROS master, which maintains the registry of all the active nodes in the net-

work. It also contains a parameter server which any node can use to store and access the global state. It acts as a lookup service for information on the nodes and the communication between them. A visual representation of a ROS parameter server can be seen below. The diagram shows that multiple nodes like camera and GNSS are running together in the case of a standard robotic system. The ROS master is aware of all the nodes and their communication. The communication between nodes takes place in two ways:

Client Based: This is similar to a client-server architecture where a node will send a request to another node for particular information. The server node will send the required information back to the client. This client-server system can be realised in two ways – services and actions –. The use of services is generally more common compared to actions. The major difference between the two is that services are synchronous, and the client node will wait for the server to send back the information before it continues its process. In contrast, in actions, the client node will not wait for the server response and continue processing.

Pub-Sub: Another method for the nodes to communicate with each other is through topics. A topic is a channel which acts as a pipe where a publisher can send information that a subscriber can access. There is no request, but new information is shared until the subscriber node has subscribed to a particular topic. This method of communication is the most common in ROS communication.

1.4 Gazebo Simulator

Gazebo is a simulator for robot research that is closely tied to ROS for simulating robots' behaviour ([11];[12]). Its main features include:

• A real-time physics engine which makes use of various technologies like ODE, Bullet, simbody and DART
• High-quality graphics making use of the OGRE engine.
• Rich set of sensors and plugins with noise generation and direct access to the Gazebo API.

The two essential executables of Gazebo are – gzserver and gzclient. The gzserver is responsible for the physics, the engine and sensor generation. The gzclient is responsible for the graphical user interface, the command line interface and the custom applications. The key elements required for a gazebo simulation are worlds, models, and plugins. The world consists of a scene, physics, models, plugins and light in a .world file using the Simulation Description Format (SDF). It consists of the environment in which Gazebo executes a particular simulation. The gzserver loads it. Models are also described in an SDF, but it includes the description of a single model like a drone. Another vital element is plugins. They define custom behavior - especially at the startup of other elements of the Gazebo simulation - and allow communication interface with external programs like ROS nodes. There are six types of plugins: world, model, sensor, visual, GUI and system, each corresponding to a particular kind of simulation element and its role. Currently, Gazebo supports plugins to be written in C++.

2 Simulation Based Verification using BDD and Gazebo

Aerial robots, like other CPS, are composed of networks of computers, sensors, and actuators. The software is usually used coupled with other software, networked sensors, and actuators, so it cannot be tested in isolation. Therefore, the testing process must enclose interaction with other components and the physical environment. Simulation-based testing utilises component and environment virtualisation [22], which enables convenient means of signal manipulation and behavior monitoring. The X-in-the Loop (XiL) testing describes different configurations for simulation-based verification. XiL environments provide interfaces to the system under test. The idea is to embed the system under test in a synthetic environment. Input vectors simulate data signals to an algorithm or a system, and the processed output values are monitored. A unique feature of XIL testing is the possibility to use the same stimulus for different configurations, and the behavior is directly comparable. Two XIL examples would be Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) testing. SiL testing executes the code within a simulated feedback loop, essentially on the development environment. It is non-real-time and targets functional verification. HiL is the testing of the executable on the target platform in a real-time setting. Beyond functional requirements, it also enables the validation of non-functional requirements. A common practice in aerial robotics is to have a dedicated flight control system and a companion computer for onboard mission management. There is a strong trend in the aerial robotics community around open-source flight control systems, particularly PX4 [23]. Its system architecture

allows a native integration to ROS using Micro Air Vehicle Link (MAVLink) [24]. PX4 further provides capabilities to make both SIL and HIL testing with Gazebo using PX4 SITL [25]. Using this infrastructure, executing the features defined in Gherkin using Behave to test ROS nodes with the flight control system and the physics-based simulation in Gazebo is possible.

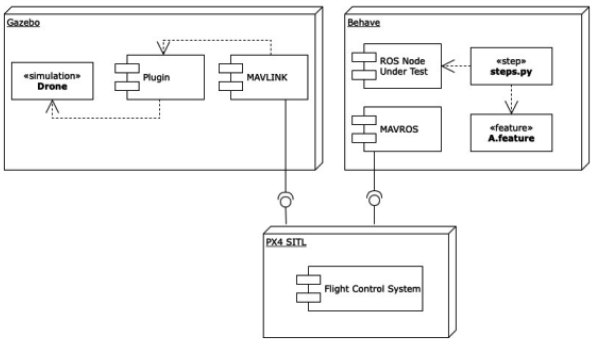

Figure 1: SITL test architecture

Figure 1 presents a simplified representation of the architecture for SIL testing. Typically, all the elements of the SIL test architecture are on the development computer and connected via UDP. PX4 SITL uses Gazebo with a plugin which provides a MAVLink connection. The specific configuration of the PX4 SITL then enables the flight control system to run in a simulation mode using firmware generated explicitly for this environment. Behave runs the step scripts executing the Gherkin feature files on the test harness side and exercises the ROS Node Under Test functions. PX4 supports the integration to ROS via MAVROS [26]. MAVROS is a ROS package that enables the connection of ROS deployments to MAVLink-enabled flight control systems, such as PX4 [27]. The test harness then utilises MAVROS to connect to the flight control system. Based on the test requirements, other ROS nodes or interactions may exist between the test harness and Gazebo. A representative case to demonstrate the utilisation of the proposed approach would be testing a ROS node whose goal is to publish data from a distance sensor. The Hokuyo sensor is used for ground proximity tracking to prevent controlled flight into the terrain. One of the requirements of the Hokuyo ROS node could be written in natural language as follows: *Hokuyo node shall provide the drone altitude above ground level when the drone is flying higher than 0.2 meters and lower than 14 meters above the ground level.* It is then possible to specify this requirement with three scenarios—examples of these scenarios as Gherkin features are in Figure 9.

2.1 Data Flow between ROS nodes

The diagram in figure 2 demonstrates the different ROS nodes, the data flow between them, and how these nodes interact to obtain the desired data. The diagram consists of four nodes:
1- ROS Master node: It represents the node responsible for registering and controlling all the other nodes to facilitate communication between them.
2- Laser scan node: This node publishes the drone altitude data received from Laser
3- GPS node: This node publishes the drone altitude data received from GPS
4- Processing node: This node subscribes to both the drone altitude data received from Laser and GPS and uses this information to control the drone accordingly.

3 Implementation steps

3.1 First step: Adding Hokuyo LIDAR sensor to the drone

To add the Hokuyo sensor to the drone [1], the following snippets have been added to the SDF file of the drone. The first is used to link it to the body of the drone, as shown in figure 3; the second shows its topic name, "/spur/laser/scan", as shown in figure 4, and the third contains its characteristics shown in figure 5.

The added sensor was rotated 90 degrees using pose coordinates so that the Hokuyo beams could be directed vertically to detect the ground, as shown in figure 6. The Hokuyo LIDAR publishes its data or distance to obstacles via a rostopic named "/spur/laser/scan" as seen in figure 4 in the topicName tag under the plugin tag.

3.2 Second step: Control the drone (Ground detection, change mode, arm, takeoff and land)

3.2.1 Ground detection

There are two ways, in our case, to detect the ground:
The Hokuyo sensor sends a range of 1024 beams (i.e. of type array) in different directions, shown in figure 6

[1] Gazebo Add a Sensor to a Robot: https://classic.gazebosim.org/tutorials

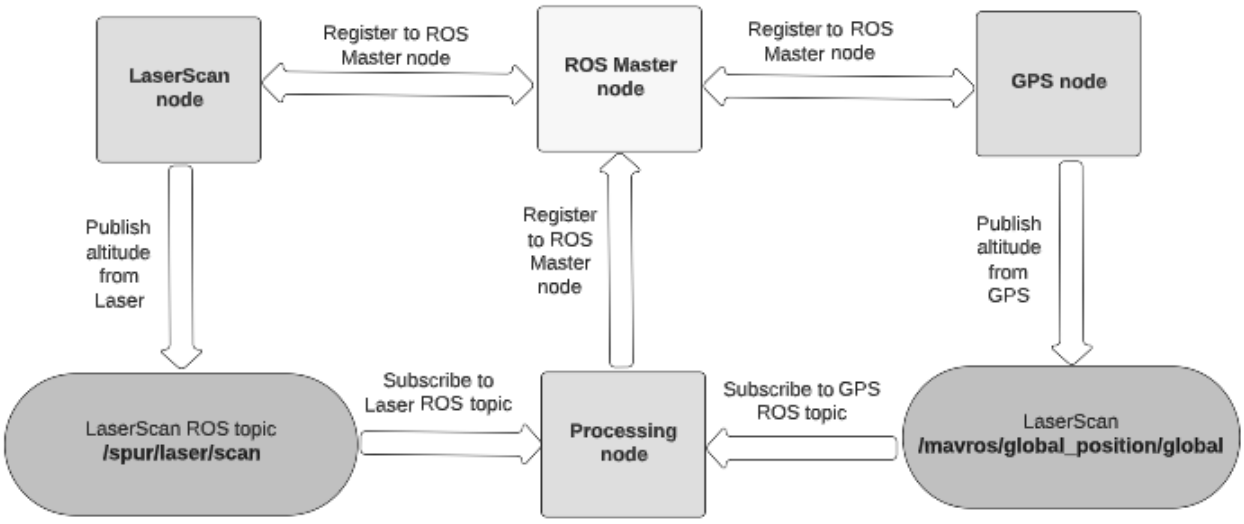

Figure 2: data flow between ROS nodes via ROS topics and services.

```
<joint name="hokuyo_joint" type="fixed">
  <pose>0 0 0.03 0 0 0</pose>
  <parent>base_link</parent>
  <child>hokuyo_link</child>
</joint>
```

Figure 3: The SDF snippet used to attach the Hokuyo LIDAR to the drone's body using the joint tag.

```
<plugin name="hokuyo_node" filename="libgazebo_ros_laser.so">
  <robotNamespace></robotNamespace>
  <topicName>/spur/laser/scan</topicName>
  <frameName>/hokuyo_sensor_link</frameName>
</plugin>
```

Figure 4: The Rostopic name through which Hokuyo publishes its data

as thin blue lines. This work aimed to detect the ground after being hit with some beams sent by the Hokuyo LIDAR.
There are two ways to perform this detection task:
• First method (using the minimum of the 1024 beams): This solution can only work if the drone is flying in a clean environment without obstacles, such as trees, buildings, or other obstacles. Suppose there are any obstacles inside the maximum range of the LIDAR; then, the minimum distance represents the distance to the closest obstacle relative to the drone. For this reason, the second method solves this problem.
• Second method (using the index of the beam directed the vertically to the ground):
After a short time, when the drone takes off, all the distances corresponding to the 1024 indexes of the array were printed to figure out which one represents the minimum, knowing there were no obstacles around to avoid mistakes in detecting other objects than the ground. After multiple tests, it was found that the index for hitting the ground vertically is index number 256.
Since the environment is clean and no obstacles exist nearby, we used the min() function for ground detection.

3.2.2 Change mode, arm, takeoff and land

The first step was to get the published distance LIDAR data sent from the sensor by subscribing to its rostopic using `rospy.Subscriber('/spur/laser/scan', LaserScan,scan_callback,queue_size=10)`
A callback function named `"scan_callback"` was defined to implement and write the code based on three conditions (figure 7):
• Distance sensor altitude of the drone less than 0,2m: Three functions corresponding to three commands are called to change the mode to offboard or guided mode, arm the drone, and then takeoff.
• Distance sensor altitude of the drone between 0,2m and 20m: In this case, the drone is considered to be

```
  <!--add lidar-->
<link name="hokuyo_link">
  <pose>0 0 0 3.14159 1.52549 3.14159</pose>
  <collision name="collision">
    <pose>0 0 0.3 0 0 0</pose>
    <geometry>
      <box>
        <size>0.1 0.1 0.1</size>
      </box>
    </geometry>
  </collision>
  <visual name="visual">
    <pose>0 0 0.27 0 0 0</pose>
    <geometry>
      <mesh>
        <uri>model://hokuyo/meshes/hokuyo.dae</uri>
      </mesh>
    </geometry>
  </visual>
  <inertial>
    <mass>0.016</mass>
    <inertia>
       <ixx>0.0001</ixx>
       <ixy>0</ixy>
       <ixz>0</ixz>
       <iyy>0.0001</iyy>
       <iyz>0</iyz>
       <izz>0.0001</izz>
       <!-- low intertia necessary to avoid not disturb the drone -->
    </inertia>
  </inertial>

  <sensor type="ray" name="laser">
    <pose>0 0 0.3 0 0 1.57</pose>
    <visualize>true</visualize>
    <update_rate>10</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>1024</samples>
          <resolution>1</resolution>
          <min_angle>-3.141593</min_angle>
          <max_angle>3.141593</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.1</min>
        <max>30</max>
        <resolution>0.1</resolution>
      </range>
    </ray>
    <plugin name="hokuyo_node" filename="libgazebo_ros_laser.so">
      <robotNamespace></robotNamespace>
      <topicName>/spur/laser/scan</topicName>
      <frameName>/hokuyo_sensor_link</frameName>
    </plugin>
  </sensor>
</link>
```

Figure 5: The SDF snippet used to add the Hokuyo LIDAR to the drone using the link tag

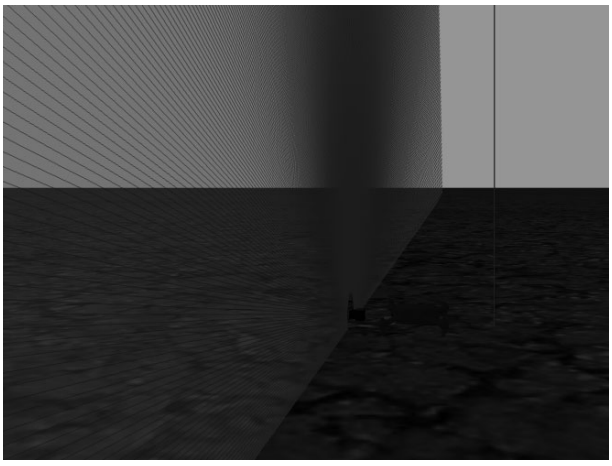

Figure 6: Hokuyo beams directed vertically to detect the ground

flying in the desired range· therefore, no action needs to be taken.
• Distance sensor altitude of the drone higher than 20m: A function corresponding to the land command is called if the received altitude is higher than the maximum altitude of 20m.

3.3 Qgroundcontrol as visualisation tool

Qgroundcontrol [2] was used to visualise the status of the drone (figure 8). It is helpful software that can be used to configure PX4 parameters and mission planning. It shows valuable information about the drone, such as altitude, location, connection status, and errors as notifications (written as well as audio notifications), which helps the user or the developer to detect and see errors and problems hidden in the code and correct them very quickly.

3.4 Third step: Gherkin scenarios in a feature file and their implementation as steps in python

As represented in figure 9, the feature file contains different scenarios; each scenario consists of three main keywords: Given, When and Then. The meaning of these are briefly explained below [3]:
• Given: It is represented by the keyword Given in the .feature file. It means under a certain situation or state
• When: It is represented by the keyword When in the .feature file. It means when a certain condition or event is met
• Then: It is represented by the keyword Then in the .feature file. It means a specific outcome must occur as a result.

3.4.1 Implementation of "Given" Hokuyo LIDAR distance sensor node running

//rework In this work, the initial situation is to check whether the Hokuyo LIDAR node is running i.e. it is publishing the LIDAR distance data. A bash script checks whether or not the LIDAR topic is publishing data(figure 11). In other words, it appears in the output of the "rostopic list" command. Searching the output of the latter command and greping the Hokuyo LIDAR

[2] QGroundControl User Guide: https://docs.qgroundcontrol.com/master/en/
[3] Gherkin Reference: https://cucumber.io/docs/gherkin/reference/

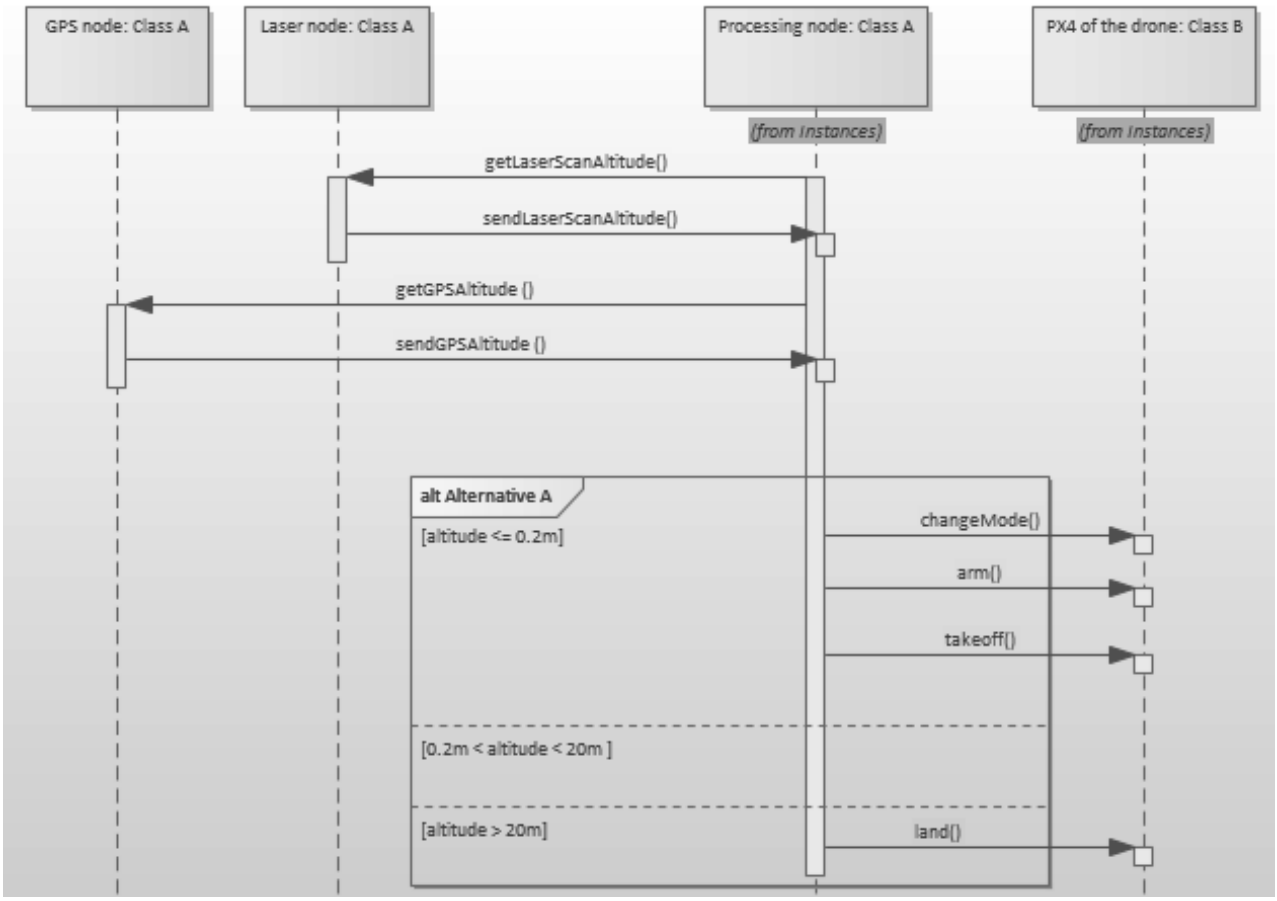

Figure 7: Sequence diagram showing the flow of commands to control the drone.

Figure 8: Picture of the QGroundControl GUI

topic name "/spur/laser/scan". If the script finds it exits successfully with 0, otherwise, it exits with another value and the checking for the node running test fails.

3.4.2 implementation of "When" events (three event possibilities) and "Then" their expected corresponding outcomes

The task was to check and test three possibilities of the drone altitude:

• First event when the drone is under 0.2m: In this case, the altitude received from the Hokuyo sensor must be minus infinity

• Second event when the drone is flying between 0.2m and 14m: In this case, the difference between the altitude received from the Hokuyo sensor and the one obtained from GPS must be less than 0.30m

• Third event when the drone is flying above 14m: In this case, the altitude received from the Hokuyo sensor must be plus infinity

```
Feature: Precision altimeter with Hokuyo sensor

Scenario: Lower than 0.2 meters AGL
Given Hokuyo sensor ROS node is running
When The drone is at an altitude less than 0.2 meters above ground level
Then Range Message of Hokuyo sensor ROS node is minus infinity.


Scenario: Lower than 14 meters and higher than 0.2 meters AGL
Given Hokuyo sensor ROS node is running
When The drone is flying lower than 14 meters and higher than 0.2 meters above ground level
Then The difference between the drone's altitude above ground level and Range Message of Hokuyo
sensor ROS node is less then 0.30 meters.


Scenario: Higher than 14 meters AGL
Given Hokuyo sensor ROS node is running
When The drone is flying higher than 14 meters above ground level
Then Range Message of Hokuyo sensor ROS node is plus infinity.
```

Figure 9: Gherkin scenarios.

4 Results and Discussion

As shown in the output of the behave tests for the 3 cases mentioned before (lower than 0,2m, between 0,2m and 14m, and higher than 14m), only one event test case should run successfully at a time. The other two test cases must fail because the drone can only be in one of the 3 cases mentioned above (figure 10).
// Rewite For the Test whether the Hokuyo node is publishing its topic, it can be seen in the output that the test runs successfully for ALL 3 cases (first line of each event test is green) regardless in which event the drone is, which is an expected result since the publication of distance topic is independent from the altitude of the drone.
For the sake of clarification, the output of the tests can be divided into three main sections:
• Given node running and publishing altitude and the drone altitude lower than 0,2m: As shown on the top of the behave test output, the corresponding event and outcome tests are in green (figure 12) and whereas the context and outcome are in red for the other 2 test cases
• Given node running and publishing altitude and the altitude of the drone between 0,2m and 14m: For the middle case of the behave test output, the corresponding event and outcome tests are in green (figure 13) and whereas the event and outcome are in red for the other 2 test cases
• Given node running and publishing altitude and the drone altitude higher than 14m: For the bottom case of the behave test output, the corresponding event and outcome tests are in green (figure 14) and whereas the event and outcome are in red for the other 2 test cases

5 Conclusion and Future Work

In conclusion, it can be said that integration of ROS, Gazebo, PX4 Autopilot, and QGroundcontrol provides an efficient way to simulate cyber-physical systems, such as the drone example we used in this project, and allows for efficient and smooth communication between these different components. Concerning the testing part, the BDD framework was a handy tool that facilitated our tests with SITL, including ROS, Gazebo and the drone PX4 autopilot firmware. It is a user-friendly framework that provides an easy-to-understand high-description language for all involved stakeholders, even those without a technical background. In this work, a Hokuyo LIDAR sensor was used, which is limited in its performance because it sends beams in all directions around it. It is suitable for object detection in general, but for detecting a specific object in a particular desired direction, like in our situation, using a directed Laser would be more efficient, which could improve the precision of the received data. Moreover, the goal of this project was mainly for learning purposes using simula-

```
OK laser scan is in rostopic list
Feature: Precision altimeter with TerraRanger One # ../drone.feature:3

  Scenario: Lower than 0.2 meters AGL                                    # ../drone.feature:5
    Given TerraRanger One ROS node is running                            # drone_steps.py:11 0.653s
    When The drone is at an altitude less than 0.2 meters above ground level # drone_steps.py:18 0.287s
    Then Range Message of TerraRanger One ROS node is minus infinity.    # drone_steps.py:25 0.098s

  Scenario: Lower than 14 meters and higher than 0.2 meters AGL
:10
    Given TerraRanger One ROS node is running
1 0.683s
    When The drone is flying lower than 14 meters and higher than 0.2 meters above ground level
2 0.100s
      Traceback (most recent call last):
        File "/home/lowe/.local/lib/python3.6/site-packages/behave/model.py", line 1329, in run
          match.run(runner.context)
        File "/home/lowe/.local/lib/python3.6/site-packages/behave/matchers.py", line 98, in run
          self.func(context, *args, **kwargs)
        File "drone_steps.py", line 37, in drone_flying_lower_thanb_14m_and_highen_than_0dot2m
          assert low_altitude_threshold < drone_terranger_altitude < high_altitude_threshold
      AssertionError

    Then The difference between the drone's altitude above ground level and Range Message of TerraRanger One ROS node is less then 0.30 meters.

  Scenario: Higher than 14 meters AGL                                  # ../drone.feature:15
    Given TerraRanger One ROS node is running                          # drone_steps.py:11 0.668s
    When The drone is flying higher than 14 meters above ground level # drone_steps.py:48 0.096s
      Traceback (most recent call last):
        File "/home/lowe/.local/lib/python3.6/site-packages/behave/model.py", line 1329, in run
          match.run(runner.context)
        File "/home/lowe/.local/lib/python3.6/site-packages/behave/matchers.py", line 98, in run
          self.func(context, *args, **kwargs)
        File "drone_steps.py", line 52, in drone_flying_higher_than_14m
          assert drone_terranger_altitude > high_altitude_threshold
      AssertionError

    Then Range Message of TerraRanger One ROS node is plus infinity.  # None


Failing scenarios:
  ../drone.feature:10  Lower than 14 meters and higher than 0.2 meters AGL
  ../drone.feature:15  Higher than 14 meters AGL

0 features passed, 1 failed, 0 skipped
1 scenario passed, 2 failed, 0 skipped
5 steps passed, 2 failed, 2 skipped, 0 undefined
Took 0m2.586s
```

Figure 10: Test output when drone altitude less than 0,2m

```
# HOKUYO LIDAR SENSOR ROSTOPIC PUBLISHING
@given('TerraRanger One ROS node is running')
def teraranger_ros_node_running(nodeCheckScriptExitValue):
    nodeCheckScriptExitValue= os.system('sh ./node_running_script.sh')
    assert nodeCheckScriptExitValue==0
```

Figure 11: Python snippet to check that Hokuyo sensor is publishing its data successfully

```
Scenario: Lower than 0.2 meters AGL
  Given TerraRanger One ROS node is running
  When The drone is at an altitude less than 0.2 meters above ground level
  Then Range Message of TerraRanger One ROS node is minus infinity.
```

Figure 12: Zoomed in output, drone altitude < 0,2m

tion in the loop (SITL). For more practical projects with the hardware (HIL), choosing a high-performance dis-

```
Scenario: Lower than 14 meters and higher than 0.2 meters AGL
                                                    # ../drone.feature:10
  Given TerraRanger One ROS node is running
                                                    # drone_steps.py:11 0.949s
  When The drone is flying lower than 14 meters and higher than 0.2 meters above ground level
                                                    # drone_steps.py:32 0.093s
  Then The difference between the drone's altitude above ground level and Range Message of TerraR
s less then 0.30 meters. # drone_steps.py:40 0.214s
```

Figure 13: Zoomed in output, 0,2m < drone altitude < 14m

```
Scenario: Higher than 14 meters AGL
  Given TerraRanger One ROS node is running
  When The drone is flying higher than 14 meters above ground level
  Then Range Message of TerraRanger One ROS node is plus infinity.
```

Figure 14: Zoomed in output, drone altitude > 14m

tance sensor like Teranger one [4] would give better and more precise results.

[4] TeraRanger One - the lightweight and low-cost ToF distance measurement sensor, 14m, 8grams: https://www.terabee.com/sensors-modules/lidar-tof-range-finders/

References

[1] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng. *"ROS: an open-source Robot Operating System"*. InICRA workshop on open source software 2009 May 12 (Vol. 3, No. 3.2, p. 5).

[2] Gupta, S., and U. Durak. 2020. *"RESTful Software Architecture for ROS-based Onboard Mission System for Drones"*. In AIAA Scitech 2020 Forum, 0239.

[3] Sankhe, C., B. Ahuja, A. Coutinho, C. Bhangale, and N. Giri. 2020 *"Implementation of ROS in Drones for Animate and Inanimate Object Detection"*. In Advanced Computing Technologies and Applications, 579–589. Springer.

[4] Honig, W., and N. Ayanian. 2017. *"Flying multiple UAVs using ROS"*. . In Robot Operating System (ROS), 83–118. Springer.

[5] Lamping, A. P., J. N. Ouwerkerk, and K. Cohen. 2018. *"Multi-UAV Control and Supervision with ROS"*. . In 2018 Aviation Technology, Integration, and Operations Conference, 4245.

[6] Fabresse, L., J. Laval, and N. Bouraqadi. 2013. *"Towards test-driven development for mobile robots"*. . In Proceedings of the ICRA 2013 Workshop on Software Development and Integration in Robotics (SDIR VIII).

[7] Dieber, B., R. White, S. Taurer, B. Breiling, G. Caiazza, H. Christensen, and A. Cortesi. 2020. *"Penetration testing ROS"*. . In Robot Operating System (ROS), 183–225. Springer.

[8] Paikan, A., S. Traversaro, F. Nori, and L. Natale. 2015. *"A generic testing framework for test driven development of robotic systems"*. . In International Workshop on Modelling and Simulation for Autonomous Systems, 216–225. Springer.

[9] Beck, K. 2003. Test-driven development: by example. Addison-Wesley Professional. Bernardeschi, C., A. Fagiolini, M. Palmieri, G. Scrima, and F. Sofia. 2018. *"Ros/gazebo-based simulation of co-operative uavs"*. . In International Conference on Modelling and Simulation for Autonomous Systesm, 321–334. Springer.

[10] Lee, E. A. 2010. "CPS foundations". In Design Automation Conference, 737–742. IEEE. Meier, L., D. Honegger, and M. Pollefeys. 2015. *"PX4: A node-based multithreaded open-source robotics framework for deeply embedded platforms"*. . In 2015 IEEE international conference on robotics and automation (ICRA), 6235–6240. IEEE.

[11] Gazebo Robotics Foundation 2014. *"Gazebo, robot simulation made easy"*. . http://gazebosim.org/. Accessed: 2021-03-01.

[12] Koenig, N., and A. Howard. 2004. *"Design and use paradigms for gazebo, an open-source multi-robot simulator"*. . In 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566), Volume 3, 2149–2154. IEEE.

[13] Takaya, K., T. Asai, V. Kroumov, and F. Smarandache. 2016.*"Simulation environment for mobile robots testing using ROS and Gazebo"*. . In 2016 20th International Conference on System Theory, Control and Computing (ICSTCC), 96–101. IEEE.

[14] Meyer, J., A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. Von Stryk. 2012. *"Comprehensive simulation of quadrotor uavs using ros and gazebo"*. . In International conference on simulation, modeling, and programming for autonomous robots, 400–411. Springer.

[15] Bernardeschi, C., A. Fagiolini, M. Palmieri, G. Scrima, and F. Sofia. 2018. *"Ros/gazebo based simulation of co-operative uavs"*. . In International Conference on Modelling and Simulation for Autonomous Systesm, 321–334. Springer.

[16] Sciortino, C., and A. Fagiolini. 2018. *"ROS/Gazebo-Based Simulation of Quadcopter Aircrafts"*. . In 2018 IEEE 4th International Forum on Research and Technology for Society and Industry (RTSI), 1–6. IEEE. Takaya, K., T. Asai, V. Kroumov, and F. Smarandache. 2016. "Simulation environment for mobile robots testing using ROS and Gazebo". In 2016 20th International Conference on System Theory, Control and Computing (ICSTCC), 96–101. IEEE.

[17] Chelimsky, D., D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. 2010. *"The RSpec Book: Behaviour Driven Development with Rspec"*. . Cucumber, and Friends, Pragmatic Bookshelf 3:25. Dieber, B., R. White, S. Taurer, B. Breiling, G. Caiazza, H. Christensen, and A. Cortesi. 2020. "Penetration testing ROS". In Robot Operating System (ROS), 183–225. Springer.

[18] Evans, E. 2004. *Domain-driven design: tackling complexity in the heart of software.* . Addison-Wesley Professional.

[19] Wynne, M., A. Hellesoy, and S. Tooke. 2017. *The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf.* .

[20] Rice, Benno and Jones, Richard and Engel, Jens 2017. *"Welcome to behave!"*. . https://behave.readthedocs.io/en/stable/. Accessed: 2021-03-01

[21] Bipin, K. 2018. *Robot Operating System Cookbook. Packt Publishing.* .

[22] Koch, C. B., U. Durak, and D. Muller. 2018. *"Simulation-based verification for parallelization of model-based ¨ applications".* . In Proceedings of the 50th Computer Simulation Conference, 1–10.

[23] Meier, L., D. Honegger, and M. Pollefeys. 2015. *"PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms".* . In 2015 IEEE international conference on robotics and automation (ICRA), 6235–6240. IEEE.

[24] Koubaa, A., A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui. 2019. *"Micro air vehicle link (mavlink) in a nutshell: A survey".* . IEEE Access 7:87658–87680.

[25] PX4 2021a. *"Gazebo Simulation".* . https://docs.px4.io/master/en/simulation/gazebo.html. Accessed: 2021- 03-15.

[26] PX4 2021b. *"ROS (Robot Operating System)".* . https://docs.px4.io/master/en/ros/. Accessed: 2021-03-15.

[27] PX4 2021c. *"ROS with MAVROS Installation Guide".* . https://docs.px4.io/master/en/ros/mavros installation.html. Accessed: 2021-03-15.