ASSIGNMENT OF BACHELOR’S THESIS

Title: State machine for drone control
Student: Matúš Gura
Supervisor: RNDr. Petr Štěpán, Ph.D.
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2019/20

Instructions
Study state machines implementation in system ROS - Robot Operating System (http://www.ros.org/).
Design and implement a state machine that uses primary drone control and detection algorithms for passing through the window.
Create a client that displays the actual state of the state machine.
Create a program that analyses log from ROS and computes the probability of transitions and states of the state machine.
Analyse created state machine in Gazebo simulation environment (http://gazebosim.org/).

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 9, 2018
Acknowledgements

I would like to express my gratitude to my advisor, RNDr. Petr Štěpán, Ph.D., for his guidance, immense patience, and knowledge. My sincere thanks also goes to my fiancé Therese, for her continuous support, and for always being there cheering me up.
I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 10, 2019
Czech Technical University in Prague
Faculty of Information Technology
© 2019 Matúš Gura. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic.
It has been submitted at Czech Technical University in Prague, Faculty of
Information Technology. The thesis is protected by the Copyright Act and its
usage without author’s permission is prohibited (with exceptions defined by the
Copyright Act).

Citation of this thesis

Technical University in Prague, Faculty of Information Technology, 2019.
Abstrakt


Klíčová slova konečný automat, dron, průlet oknem, ROS, SMACH

Abstract

This thesis implements a state machine that navigates a drone through an open window. It uses Robot Operating System as the primary toolset for the implementation. The system’s concepts are studied thoroughly. The modular and simple solution that was achieved is capable of realtime visualisation. Additionally, a program that analyses past executions of the solution is presented.

Keywords state machine, drone, passing through a window, ROS, SMACH
# Contents

## Introduction
- 1 Definitions
  - 1.1 Drone
  - 1.2 State Machine
  - 1.3 Robot Operating System
  - 1.4 Gazebo

## Current State
- 2.1 Drone
- 2.2 Simulation
- 2.3 Available Communication

## Analysis and Design
- 3.1 Requirements
- 3.2 Programming Language Selection
- 3.3 State Machine Implementation Analysis
- 3.4 SMACH
- 3.5 State Machine Execution Analyser

## Development
- 4.1 Package Overview
- 4.2 Simulating Window Detection
- 4.3 State Machine Implementation
- 4.4 Analyser Implementation

## Testing
- 5.1 Gazebo Simulation
- 5.2 State Machine Viewer
- 5.3 State Machine Analyser
# List of Figures

1.1 Aircraft principal axes of an airplane from [2] ............... 4  
1.2 State diagram of a finite state machine accepting numbers divisible  
    by three .................................................. 6  
1.3 Visualisation of ROS communication models .................... 7  
2.1 The used drone from MRS .................................. 11  
2.2 Intel® RealSense™ Depth Camera D435 ....................... 12  
2.3 Provided world simulation ................................. 13  
4.1 State machine implementation .............................. 23  
5.1 SMACH Viewer with the current state highlighted ........... 28
List of Tables

5.1 State transition analysis of the implemented state machine . . . . 29
Introduction

Humankind had dreamt of flying with birds in the sky since the earliest days. Men have been building the first flying apparatuses since before the common era. Because of the motivation to fly like birds, the word “aviation” from the Latin word “avis” which means “bird”, came into light.

In the current aviation generation, people are familiar with aviation vehicles which provide excellent control, stability, reliable power, lift and efficiency. It is commonly used for long-distance travelling or for military purposes.

Due to the world’s ever-growing technological advances, the world was introduced to Unmanned Aerial Vehicles (UAVs) now typically called “drones”. As how the name suggests, UAVs are aviation vehicles wherein a human operates the vehicles remotely or vehicles are fully autonomous.

Nowadays, big technology companies compete in drone research and its application in real-life scenarios. They are used for deliveries, commercial aerial surveillance, even disaster relief. However, let us not forget that drones are also used for leisure. Flight is something that has fascinated the humankind for years so it’s no wonder why enthusiasts fiddle with these flying machines.

As a software engineer, I am interested in drones as a tool to ease human processes. I am confident that humanity will focus more on the creative parts of jobs wherein most tasks can be automated - including driving any mode of transportation. With this in mind, this thesis is aimed to be remembered as a resource for drone automation. It seeks to aid the Czech Technical University’s Multi-Robot Systems group (MRS) knowledge in future competitions, research studies or personal information.

The main goal of this thesis is to design and implement a state machine that uses primary drone control and detection algorithms for passing through a window. Main definitions and relevant terms are introduced in Chapter 1. In Chapter 2, provided resources are studied. Chapter 3 analyses available implementation options to develop a state machine for drone control and a tool to analyse its execution. The implemented solution is described in Chapter 4, and it is evaluated in Chapter 5.
CHAPTER 1

Definitions

The aim of this chapter is to provide an overview of several important definitions needed while reading this thesis.

1.1 Drone

A drone is defined as an unmanned vehicle. It consists of

- a *platform* - a drone itself,
- a *payload* - equipment attached to a drone.

This distinction allows to accommodate different goals of a drone by changing a combination of a payload it carries.

We limit our discussion to an unmanned aerial vehicle (UAV). It is defined as “a pilotless aircraft (…), which is flown without a pilot-in-command on-board and is either remotely and fully controlled from another place (ground, another aircraft space) or programmed to be autonomous” [1].

1.1.1 Aircraft Principal Axes

The orientation of a drone in 3D space is defined by three principal axes for rotation:

- **Yaw** (Normal Axis) - This axis advances an aircraft in rotations such as clockwise and counterclockwise. A positive yaw value moves the nose of an aircraft clockwise.

- **Pitch** (Transverse Axis) - This axis advances an aircraft on the side axis, allowing the aircraft to tilt up and down from front to back. A positive pitch value lifts the head and lowers the tail.
1. Definitions

- **Roll** (Longitudinal Axis)- This axis enables the aircraft to lean from side to side. A positive roll value moves the left wing up while lowers the right wing.

![Figure 1.1: Aircraft principal axes of an airplane from [2]](image)

1.1.2 Velocity

The movement of a drone can be defined by two independent values for speed. The measures can be observed given an object moving in a circle around a fixed centre point with a constant speed:

- **angular velocity** is a measure of the object rotation in radians per time unit,

- **linear velocity** is a measure of distance the object travelled per time unit.

1.2 State Machine

A state machine (shorter for finite-state machine) is an abstract machine designed to accept or reject an input based on the given pattern. The given pattern is called a transition function. The state machine internally holds a finite set of states. The state can change to another in response to a given input. This change is called a **transition**.

Two types of finite state machines can be distinguished:

- deterministic finite state machines,
1.2. State Machine

- nondeterministic finite state machines.

A deterministic finite state machine $D$ is a quintuple

$$D = (Q, \Sigma, q_0, F, \delta)$$

where

- $Q$ is the finite, non-empty set of states,
- $\Sigma$ is the finite, non-empty set of symbols (letters),
- $q_0$ is the initial state ($q_0 \in Q$),
- $F$ is the set of final (accepting) states ($F \subseteq Q$),
- $\delta$ is the state transition function $\delta : Q \times \Sigma \rightarrow S$.

A nondeterministic finite state machine is the same quintuple as a deterministic finite state machine, but the state transition is defined as $Q \times \Sigma \rightarrow P(Q)$. However, it is not needed for the purposes of this thesis.

1.2.1 State Diagram

The machine can be visualised in a diagram called state diagram. It is a diagram where:

- states are drawn as labeled circles,
- transitions are drawn as directed edges connecting the states labeled with the input symbol causing the transition,
- a start state is drawn as an arrow with no origin state,
- an accepting state is drawn as a double circle.

1.2.2 Hierarchical Finite State Machine

A hierarchical state machine is an extension of finite state machines. It was first introduced by D. Hare in [3] under the name ”StateCharts”. It introduces super-states that contain other states. A super-state can have transitions that are called generalized transitions.
1. Definitions

Figure 1.2: State diagram of a finite state machine accepting numbers divisible by three

1.3 Robot Operating System

The Robot Operating System (ROS) is an open source collection of tools for writing robot software. The main motivation behind creating such a collection is the ability to reuse code in robotics, ultimately accelerating robot development. To achieve such an ambitious goal, “ROS was designed to be as distributed and modular as possible” [4].

In this section, we introduce the complex modular architecture of ROS and some of its tools that would enable us to use and extend ROS.

1.3.1 Concepts

ROS system consists of numerous simultaneously running computer programs while communicating with each other.

These programs are called nodes. In practice, there can be a node to control motors, a node to capture images, and another to listen to an input device. This separation into executables exists to provide fault tolerance and more straightforward code separation. Nodes must have unique names that are used to communicate with other nodes by sending messages. A message is a strictly typed and language agnostic data structure [5].

ROS offers two communication models:

1. publish-subscribe model using topics,

2. request-response model using services.

Topics are streams of data with unique names that receive messages from a publishing node and send them to all subscribed nodes. The data can either be transmitted using default TCP/IP or UDP connections.
1.3. Robot Operating System

Services are used when a node needs to receive a reply. Any node can offer a service. However, ROS does not implement any standard services.

![Visualisation of ROS communication models](image)

Figure 1.3: Visualisation of ROS communication models

The whole architecture is orchestrated by one central *master* process called *roscore*. When a node is started, it has to register itself with the master. The master then provides the node with relevant connection information needed to form a direct connection with other nodes. This means that messages are not transmitted through the master. It only serves as a lookup mechanism to allow processes to find each other (very much like a DNS server). A node creation, a node termination, and a connection between two nodes happen dynamically.

### 1.3.2 File system

ROS code is stored in the set of directories inside a directory called a *workspace*. There can be only one active workspace at the time.

A workspace contains *packages*. A package is the main software unit of the ROS system that can contain multiple nodes. It has to meet the following requirements:

1. It contains an XML manifest file named *package.xml*. The file defines properties such as the package name, license, authors, dependencies etc. It is important when sharing the package.

2. It contains an input file to the CMake build system named *CMakeLists.txt*. The overall structure is described in [6].

3. It has its own folder (no nested packages are allowed).

### 1.3.3 Messages

Messages are a crucial concept of ROS since they are used to wrap all transferred data. ROS comes with built-in primitive message types and additional common types to improve interoperability throughout the system.
1. Definitions

It also offers to define custom messages by creating a simple text file in msg directory of a package. Every line of a msg file starts with a field type (any existing message type present in the system), follows by a space, and ends with the field name.

Some of the primitive ROS messages that are being used in this thesis follow:

- **bool** represents a boolean value,
- **int32** represents a signed 32-bit integer,
- **uint32** represents an unsigned 32-bit integer,
- **float32** is a 32-bit IEEE float number,
- **float64** is a 64-bit IEEE float number,
- **string** represents an ASCII string.

This section further describes some of the common geometric primitives with their compact message definitions.

1.3.3.1 geometry_msgs/Point

Point represents a point in space. It has three components:

    float64 x
    float64 y
    float64 z

1.3.3.2 geometry_msgs/Quaternion

Quaternion represents an orientation in space. It is widely used to track and apply rotations in ROS. It has four components:

    float64 x
    float64 y
    float64 z
    float64 w

1.3.3.3 geometry_msgs/Pose

Pose represents a pose of an object in space. It is composed of position in space and its orientation:

    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
1.3. Robotics Operating System

1.3.3.4 geometry_msgs/Vector3

Vector3 represents a vector in space with three components:

```plaintext
float64 x
float64 y
float64 z
```

It only represents a direction. Previously mentioned Pose is more suitable when vector transformations are needed.

1.3.3.5 geometry_msgs/Twist

Twist contains angular and linear velocity of an object:

```plaintext
gometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

1.3.4 Logging

ROS offers a rosout package that allows to log custom messages at different verbosity levels.

Furthermore, logging can be done using bags. A bag is a file format for storing ROS messages that represent the same structure used in the network transport layer of ROS. This representation allows for efficient recording and playback of messages.

1.3.5 Building

The official set of tools for building ROS code (build system) is catkin. It is included by default in ROS installation package.

Catkin extends CMake to manage dependencies between packages [7], but it is ROS independent.

1.3.6 Commands

This section describes commands needed to control ROS system:

- **roscore** - command to start the ROS master and other system necessary programs,
- **rosrun** - command to locate a package in the current workspace and executes it with given arguments,
- **rostopic** - command to list all active topics and print/publish their messages,
1. Definitions

- **rosservice** - command to list all active services and call them with the provided arguments,
- **rosmsg** - command to list all messages,
- **rosbag** - command to record and replay a bag file,
- **catkin** - command to build ROS packages,
- **catkin_create_pkg** - command to create a ROS package.

1.4 Gazebo

A simulator will be the primary validation tool of coordination mechanisms discussed in this thesis. Although ROS supports multiple simulators, the thesis will only focus on the use of Gazebo simulator as an assigned choice.

A simulation in Gazebo simulates a **world**. A world is a collection of all the **objects** in the simulation, their physical parameters and global parameters of the simulation.
This chapter describes the status of the project about to be extended provided by Multi-Robot System Group from Faculty of Electrical Engineering Czech Technical University in Prague.

2.1 Drone

The main structure of the assumed drone consists of:

- a DJI F550 frame with six arms;
- six DJI E310 motors;
- a personal computer Intel® NUC with Intel® i7 processor;
- a PixHawk flight controller.

Figure 2.1: The used drone from MRS
2. Current State

The personal computer onboard takes care of drone coordination, state estimation, and motion planning for complex missions. The PixHawk flight controller contains a set of sensors such as accelerometers, gyroscopes, and magnetometers, that are necessary for a stable flight. Transport of messages between the onboard PC and PixHawk autopilot is performed over a serial line using MAVlink protocol.

The Intel® RealSense™ Depth Camera D435 is used to detect the window. With the global image shutter and wide field of view, the camera offers accurate depth perception when the object is moving or when the device is in motion. It also covers more field of view that helps avoid blind spots. The camera is using a powerful vision processor supporting up to five MIPI Camera Serial Interface 2 lanes to compute real-time depth images and accelerate output [8]. A new and advanced stereo depth algorithm is used for more accurate and extended range depth perception.

Figure 2.2: Intel® RealSense™ Depth Camera D435

For the purposes of this thesis, it is assumed that the back of the drone is between its two red arms. The front of the drone is therefore at the opposite side. It is assumed that the camera is placed above the drone facing its front.

2.2 Simulation

The project to be extended is provided as a Gazebo simulation. This section describes the simulated environment.

2.2.1 Used Software

The following software are required to run the given simulation:

- operating system Ubuntu 18.04 LTS,
2.2. Simulation

- ROS Melodic,
- Gazebo 9.0.

There are no clear minimal system requirements due to the distributed nature of ROS system. However, a multi-core processor with at least 8 GB of RAM is recommended. Insufficient computing power would most likely cause the simulation to misbehave and crash the drone.

2.2.2 Models

The world definition used in the simulation (shown in Figure 2.3) contains three models:

- ground\_plane,
- brick\_building,
- a drone.

The model brick\_building is a 12 meters tall building with square foundation of 8 meters. It has four floors and every floor has four windows at each side.

A spawned drone is identified with environment variable UAV\_NAME, typically uav1.

![Figure 2.3: Provided world simulation](image)
2. Current State

2.3 Available Communication

This section describes topics and services that are available in the simulation. Out of all the options studied only actions that are used in this thesis are described.

2.3.1 Take Off Procedure

The take off procedure consists of the following actions executed sequentially:

$ rosservice call /$UAV_NAME/mavros/cmd/arming 1;
$ rosservice call /$UAV_NAME/mavros/set_mode 0 offboard;
$ rosservice call /$UAV_NAME/control_manager/motors 1;
$ rosservice call /$UAV_NAME/mav_manager/takeoff;

2.3.2 Topics

The following topics are provided in the simulation:

- /gazebo/model_states - topic published by Gazebo simulator that contains geometry_msgs/Pose and geometry_msgs/Twist data of all objects in the simulated world,
- /$UAV_NAME/mavros/local_position/pose - topic that publishes drone’s geometry_msgs/Pose data.

2.3.3 Services

The following services are provided in the simulation:

- /$UAV_NAME/control_manager/goto_relative - a service that controls the movement of a drone from the drone’s perspective,
- /$UAV_NAME/control_manager/set_yaw_relative - accepts radians to adjust yaw of a drone and turn it clockwise or counter clockwise.
Analysis and Design

Since the outcome of this thesis is suppose to be shared with other users, it has to be built as a custom ROS package.

3.1 Requirements

The outcome of this thesis is a software program that implements a state machine. The functional requirements of such state machine are:

- navigation of a drone to a location received from an external window detection program,
- recording of every state transition to allow further analysis of the executed behaviour,
- real-time visualisation in the form of a state diagram with the current state highlighted.

The developed solution has to meet the following non-functional requirements:

- portability between systems that fulfil requirements in Section 2.2.1,
- use of ROS system and its concepts.

3.2 Programming Language Selection

ROS currently focuses on providing full non-experimental support in two programming languages: C++ and Python. This support is provided via client libraries that allow accessing its concepts through simple application programming interface (API) calls.

The C++ client library roscpp is “designed to be the high-performance library for ROS”.
3. Analysis and Design

The Python client library *rospy* enables fast prototyping due to the interpreted nature of Python programming language. However, this doesn’t mean it is not fast enough since even the ROS master service *roscore* is implemented purely using Python client library.

The state machine does not necessarily require high-performance. On the other hand, the fast prototyping would be a great benefit. This characteristic leads to the conclusion that Python is the most appropriate choice as the programming language for the purposes of this thesis.

3.3 State Machine Implementation Analysis

There are a number of ways on how to implement a state machine in Python. Three options were considered:

1. custom Python implementation,
2. implementation using an existing Python library,
3. implementation using an existing ROS package.

The findings are discussed in this section.

3.3.1 Custom Implementation

**Advantage:** This option is the most flexible as it is not bounded or limited by any existing code and it could be fitted perfectly for the purposes of this thesis.

**Disadvantages:** This implementation would require heavy time investment, especially for the introspection. Also, it would be more prone to bugs and would require deep testing as the program would be written from scratch and we would most probably be its only users.

3.3.2 Use Of Existing Python Library

The most popular Git repository hosting service called GitHub was used to search for the highest rated Python library related to state machines. Search results found that a Python library *transitions* [9] fits the criteria.

**Advantages:** The library displays itself as a strong contender as it appears to be well tested with unit tests. Moreover, it has a pool of active community of users that reaffirms its readiness for use in a production environment.

**Disadvantages:** A real-time introspection does not exist. However, a static state diagram generator is available via an extension.
3.3.3 Use Of Existing ROS Package

**Advantages:** SMACH is a Python library that has a direct ROS binding. It is well-tested by a pool of community users. It includes introspection viewer that enables a visualization of the real-time state machine while highlighting the current state.

**Disadvantages:** It is not actively developed anymore. However, it is still supported.

3.3.4 Conclusion

SMACH is the most appropriate choice for this thesis since it supports a real-time introspection and it contains direct ROS binding.

3.4 SMACH

SMACH is a Python library based on hierarchical state machines. At its core, it is a ROS independent library. However, a package `smach_ros` exists that contains ROS bindings.

This section describes how to implement a state machine using SMACH.

3.4.1 State Machine

A state machine is a container that holds states and relationships between them. It cannot be used for dynamic behaviour as the states representing actions are linked statically and cannot be freely ordered at runtime.

An example of a basic state machine implementing a light switch is shown in Listing 3.1.

```python
import smach
sm = smach.StateMachine(outcomes=["error"])
with sm:
    smach.StateMachine.add("ON", On(),
                        transitions={"switch_pressed": "OFF",
                                     "switch_broke": "error"})
    smach.StateMachine.add("OFF", Off(),
                        transitions={"switch_pressed": "ON",
                                     "switch_broke": "error"})
sm.execute()
```

Code Listing 3.1: Sample SMACH state machine

A state machine is constructed with `smach.StateMachine()` that accepts `outcomes` - a list of strings representing possible final states of the state machine.
3. Analysis and Design

States are added to the state machine with the following required parameters:

- **label** - a string,
- **state instance** - any class implementing `smach.State` interface,
- **transitions** - a dictionary with keys representing edge labels and the values representing next state or state machine outcome.

A state machine is started when a method `execute()` is called.

### 3.4.2 States

A state in SMACH corresponds to a task being performed. This differs from the formal state machine definition, where a state is a configuration of the system rather than the execution of an action.

Each state has a list of potential outcomes defined in the class constructor method `__init__()`. This method initializes the state class. It should never contain any blocking code in the main thread.

The actual work is done in method `execute()` that is called when a state machine moves into the state. The current state is considered finished once a value is returned. The returned value is an outcome that leads the state machine to the next state according to the statically linked states graph.

A basic example of a state implementation in SMACH is shown in Listing 3.2.

```python
class ExampleState(smach.State):
    def __init__(self, outcomes=['outcome1', 'outcome2']):
        self.num = random.randint(1,2)

    def execute(self):
        if self.num == 1:
            return "outcome1"
        return "outcome2"
```

Code Listing 3.2: Sample SMACH state

SMACH allows to pass data from one state to the next state. It is done by providing a list of `input_keys` and a list of `output_keys` in the state constructor, as shown in Listing 3.3. The state expects these values to be present in `userdata` structure that is passed as an argument into `execute()` method. Values listed in `input_keys` are immutable.

```python
class ExampleState(smach.State):
    def __init__(self, outcomes=['outcome1', 'outcome2'],
                 input_keys=["input_value"],
                 output_keys=["output_value"]):

    def execute(self, userdata):
```

18
Data passed between states have to be connected at the state machine level as shown in Listing 3.4.

Code Listing 3.4: Sample SMACH state machine with input and output data

3.4.3 Introspection

SMACH offers a graphical representation of a state machine called SMACH Viewer. It can visualize transitions between states, the currently active state, and the values of user data.

A state machine has to run an introspection server to use the viewer. This is done by importing smach_ros Python package and adding the code shown in Listing 3.5.

Code Listing 3.5: Introspection server in SMACH
3. Analysis and Design

3.5 State Machine Execution Analyser

The goal of this section is to discuss how to analyse state transitions of the implemented state machine.

The analyser is expected to read data from a log file. It doesn’t need access to any of ROS concepts. Therefore, it should not be implemented as a ROS package.

The problem of such analyser can be generalised into a problem of state transition logging during the state machine execution, because the parsing of log files depends on our storage structure.

As mentioned in Section 1.3.4, there are two methods to log data in ROS:

- native ROS logging using `rospy` Python package,
- logging messages into bags using `rosbag` tool.

3.5.1 Native ROS Logging

ROS provides several native methods to log string messages depending on their verbosity into several target locations, including a text file.

Since SMACH already logs every state transition, this logging method would not require any changes in the state implementation itself. However, stored log messages are strings, and their structure depends on the local environment configuration. A change in the message structure would break the parsing implementation, and a high dependency on environment configuration would cause issues when collaborating in a team.

3.5.2 Logging Into Bags

Another logging method is to create a custom message that can be stored in a bag file. A bag file maintains the message structure, therefore, no additional parsing is needed.

SMACH doesn’t broadcast any state transition messages. However, a SMACH state machine instance provides a method that registers a callback for the state machine start, a state transition and the state machine termination. This can be used for publishing a custom message.

3.5.3 Conclusion

Given the reasoning above and given the non-functional requirements of the developed solution, logging a custom message into a bag file is the chosen implementation.
This chapter explains the implementation of the state machine that navigates a drone through a window. It is assumed that a drone is already spawned in simulation and its name is stored in the environment variable `UAV_NAME`.

### 4.1 Package Overview

The implementation is dependent on the following packages:

- **smach** - ROS independent state machine implementation,
- **smach_ros** - an interface between SMACH and ROS,
- **rosbag** - a library to work with ROS bags,

The implementation is split into the following components:

- **window_locator** - a package that broadcasts a window location relative to a flying drone,
- **window_automaton** - a state machine implementation,
- **window_analyser** - a script that analyses a bag file with the state machine transitions data.

### 4.2 Simulating Window Detection

For testing and validation purposes a program broadcasting a simulated output from a primary drone control and detection algorithms is necessary to be implemented. Since it won’t be used in the production environment where it will be replaced by an actual window detection program, it is separated into its own ROS package.
4. Development

The package is named \texttt{window\_locator}. It broadcasts a window location relative to a drone via a topic \texttt{/UAV\_NAME/window\_locator/location}. The topic broadcasts a custom message \texttt{WindowLocation}.

4.2.1 \texttt{WindowLocation} Message

The message is defined as follows:

\begin{verbatim}
float32 distance
float32 roll
float32 pitch
float32 yaw
\end{verbatim}

4.2.2 Implementation

The program subscribes to \texttt{/gazebo/model\_states} topic that is described in Section 2.3.2. On every message received from the topic, a new \texttt{WindowLocation} message is created and published into \texttt{/UAV\_NAME/window\_locator/location} topic. The target window position is hard-coded with coordinates \(x_w = 0\), \(y_w = 4\) and \(z_w = 4.5\). The current drone coordinates in the world are noted as \(x_d\), \(y_d\) and \(z_d\).

A \textit{distance} between a drone and the window is computed using the 3D analogy of the Pythagorean Theorem:

\[
distance = \sqrt{(x_d - x_w)^2 + (y_d - y_w)^2 + (z_d - z_w)^2}.
\]

Computing the angle between the drone and the window means computing the relative rotation between two quaternions. This is defined as a product of the first quaternion and the inverted second quaternion. The result is then converted into Euler angles and \textit{yaw} is used.

A \textit{roll} is simply an opposite of the current roll of the drone while a \textit{pitch} is computed as

\[
pitch = \arcsin((z_w - z_d)/distance).
\]

4.3 State Machine Implementation

This section describes the states of the implemented state machine. The state diagram of the developed state machine is shown in Figure 4.1.

Every state implements either a simple timeout or a waiting mechanism that subscribes to \texttt{/UAV\_NAME/mavros/local\_position/pose} topic to wait for the drone to take the requested position. If there is an exception raised while executing a state, the outcome of the state is set to "failed", and the state machine ends with failure.
4.3. State Machine Implementation

4.3.1 TAKE_OFF State
The state executes the exact procedure described in Section 2.3.1. If the take off procedure finishes without any issue, the state outcome is set to "took_off" and the state machine moves to CHECK_FOR_WINDOW state.

4.3.2 CHECK FOR WINDOW State
The state subscribes to /$UAV_NAME/window_locator/location topic and waits for five seconds. If a message from the topic is received in the given time, "window_found" is returned causing the state machine to transition to
FACE_WINDOW state. If no message is received, the state sets user variable rotation to 0 and returns "no_window". This triggers the state machine transition to ROTATE state.

4.3.3 ROTATE State

This state rotates a drone around the yaw axis by 45 degrees if the drone hasn’t completed a full circle yet. A variable rotation representing number of the state executions is received. If its value is greater than 8, the state returns "finished_360", and the state machine ends. Otherwise, the state:

1. increases the counter,
2. sends a message into /$UAV_NAME/control_manager/set_yaw_relative service to rotate the drone by 45 degrees,
3. waits for the rotation to finish,
4. triggers the state machine transition to CHECK_FOR_WINDOW state by returning "rotated" outcome.

4.3.4 FACE_WINDOW State

The state makes sure that a drone is directly facing the window. It rotates the drone using /$UAV_NAME/control_manager/set_yaw_relative service. Once the rotation is finished, the state returns "succeeded" and the state machine transitions to ADJUST_ALTITUDE state.

The difference between ROTATE state and FACE_WINDOW state is that ROTATE state is only executed before the window is detected while FACE_WINDOW state is executed with the window being somewhere in front of the drone.

4.3.5 ADJUST_ALTITUDE State

The state makes sure that a drone and the window are at the same altitude. It does so by reading z coordinate of the window and navigating the drone to the same altitude using /$UAV_NAME/control_manager/goto_relative service. Once the action is finished, the state returns "succeeded" and the state machine transitions to GO_FORWARD state.

4.3.6 GO_FORWARD State

The state navigates a drone one meter forward by sending a request into /$UAV_NAME/control_manager/goto_relative service. Once the action is finished, the state returns "succeeded" and the state machine transitions to WAIT_FOR_BALANCE state. However, if the distance from the window is lower or equal to 1 meter, the returned outcome is "finished". This outcome triggers the state machine to end with success.
4.4. Analyser Implementation

4.3.7 WAIT_FOR_BALANCE State

One of the most significant problems during the development was the drone getting out of control and crashing into the ground. The reason for this behaviour was later identified as not enough computing power. However, this state is implemented to wait for the drone stabilization. The state might be unnecessary for the real environment.

Once the drone’s pose doesn’t significantly change in one second, the state returns succeeded. This causes the state machine to transition to CHECK_FOR_WINDOW state.

4.4 Analyser Implementation

As concluded in Section 3.5, a new custom message has to be added into the state machine implementation to log its state transitions into a bag file. Also, another Python script has to be created to open and parse the bag file and print the time spent in each of the states into the standard output.

4.4.1 AutomatonTransition Message

The custom message is named AutomatonTransition, and it is formally defined as:

```
string state
string action
```

The action component can have three possible values: "start", "transition" and "termination". If its value is either "start" or "transition", the value of state is the next state. If its value is "termination", the value of state is the outcome of the state machine.

4.4.2 Discussion

The state machine implementation has to be extended with three callbacks that publish AutomatonTransition message into /window_automaton topic, as shown in Listing 4.1.

```python
publisher = rospy.Publisher("/window_automaton",
                                  AutomatonTransition,
                                  queue_size=10)

def start_cb(userdata, next_states):
    publisher.publish(AutomatonTransition(next_states[0],
                                          "start"))

def transition_cb(userdata, next_states):
    publisher.publish(AutomatonTransition(next_states[0],
                                          "transition"))
```
def termination_cb(userdata, terminal_states, outcome):
    publisher.publish(AutomatonTransition(outcome, "termination"))

sm = smach.StateMachine()

# State machine definition
sm.register_start_cb(start_cb)
sm.register_transition_cb(transition_cb)
sm.register_termination_cb(termination_cb)
sm.execute()

Code Listing 4.1: SMACH callbacks

The Python program that analyses the bag file is named `window_analyser`. It opens a bag file, filters only messages from `/window_automaton` topic, computes time spent in each of the states of the state machine, and prints the result into the standard output.
5. Testing

This chapter describes how to run and test the implemented programs and validate their results.

The environment preparation is described in Appendix B, and the process to start Gazebo simulation is described in Appendix C. It is assumed that the provided simulation is running, both `window_locator` and `window_automaton` are present in the current ROS workspace, and all packages are built.

5.1 Gazebo Simulation

The state machine is tested in the provided Gazebo simulation.

Both implemented ROS packages have to be started in separate bash terminals in the following order:

1. a window detection package to broadcast window position relative to a drone starts by executing:

   ```
   UAV_NAME=uav1 rosrun window_locator index.py
   ```

2. the state machine starts by executing:

   ```
   UAV_NAME=uav1 rosrun window_automaton index.py
   ```

A drone takes off, and it successfully navigates through the window of the simulated building. However, two issues have been observed:

- a drone might misbehave, lose balance and fall due to the insufficient computing power of the testing device,
- the state machine causes a drone to crash shortly after passing through a window.

The latter issue is considered to be out of the scope of this thesis.
5. Testing

5.2 State Machine Viewer

When the state machine is running, we can visualise its states and its transitions using SMACH Viewer. The viewer starts by executing:

```
rosrun smach_viewer smach_viewer.py
```

It visualises the state diagram of the state machine and it highlights the current state, as shown Figure 5.1.

![SMACH Viewer with the current state highlighted](image.png)

Figure 5.1: SMACH Viewer with the current state highlighted
5.3 State Machine Analyser

A log file has to be created first to analyse the state machine execution. It is created by recording messages into a bag file using `rosbag` command:

```
rosbag record -a
```

The above command records all messages from all topics registered in the ROS master and it generates a file in the current working directory. Its output can be analysed with `window_analyser` script by running:

```
python ./window_analyser.py LOG_FILE
```

The analysis of the implemented state machine showing the average time spent in each of the states during the three simulations is shown in Table 5.1.

Table 5.1: State transition analysis of the implemented state machine

<table>
<thead>
<tr>
<th>State</th>
<th>Average Duration [s]</th>
<th>Average Duration [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAKE-OFF</td>
<td>10.05</td>
<td>8.66</td>
</tr>
<tr>
<td>CHECK_FOR_WINDOW</td>
<td>9.81</td>
<td>8.45</td>
</tr>
<tr>
<td>ROTATE</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>FACE_WINDOW</td>
<td>13.14</td>
<td>11.31</td>
</tr>
<tr>
<td>ADJUST_ALTITUDE</td>
<td>34.51</td>
<td>29.72</td>
</tr>
<tr>
<td>GO_FORWARD</td>
<td>15.76</td>
<td>13.57</td>
</tr>
<tr>
<td>WAIT_FOR_BALANCE</td>
<td>32.85</td>
<td>28.29</td>
</tr>
<tr>
<td>Sum</td>
<td>116.12</td>
<td>100.00</td>
</tr>
</tbody>
</table>
The thesis analysed and implemented a state machine in Robot Operating System to navigate a drone through a window. The process was simulated in Gazebo simulation that was provided by Multi-Robot System group at Czech Technical University in Prague. Robot Operating System was extensively studied.

The final state machine can be viewed at runtime in a client that displays the current state. It can also be analysed from a log file. The analysis computes time spent in each of the states. I believe the presented solution is a good resource for implementing state machines in Robot Operating System.

However, the current state machine implementation does not account for communication problems (communication silence) which results in drone rotating. This could be improved in the future. In regards to a drone passing through a window, behaviour trees could be researched as an interesting alternative of state machines.
Bibliography


Acronyms

API  Application Programming Interface
ASCII American Standard Code for Information Interchange
DNS  Domain Name System
IEEE Institute of Electrical and Electronics Engineers
MRS  Multi-Robot Systems
ROS  Robot Operating System
TCP/IP Transmission Control Protocol / Internet Protocol
UAV  Unmanned Aerial Vehicle
UDP  User Datagram Protocol
XML  Extensible Markup Language
Preparing Environment

The required software is described in Section 2.2.1. ROS can be installed by following [10] or using script `./lib/install_ros.sh` in the enclosed flash drive.

B.1 Provided Packages

The provided packages are located in `./lib` and `./src` directories in the enclosed flash drive. Copy all files from both of these directories into your current workspace. The origin of source code files in `./lib` directory is noted in `./lib/README.md` file.

B.2 Compiling a Workspace

To compile all packages in a workspace, simply navigate anywhere into the workspace and run:

```bash
catkin build [package_name]
```

To clean the workspace, run:

```bash
catkin clean [package_name]
```
All the following commands are meant to be executed each in a new bash terminal. They start `fire_simulation` in Gazebo with one drone:

1. Make sure that all files from the enclosed flash drive were copied into the current ROS workspace.
2. Run the ROS master process:
   
   ```bash
   roscore
   ```

3. Navigate to the current ROS workspace and build all ROS as described in Section B.2.
4. Run the Gazebo simulator by executing:
   
   ```bash
   waitForRos; \
   roslaunch fire_simulation simulation.launch gui:=true
   ```

5. Spawn a new drone by executing:
   
   ```bash
   waitForSimulation; \
   spawn 1 --run --delete --enable-rangefinder \ 
   --enable-ground-truth
   ```

6. Run `mrs_mav_manager` node:
   
   ```bash
   export UAV_NAME=uav1; \
   waitForOdometry; \
   roslaunch mrs_mav_manager simulation.launch
   ```

To execute the implemented state machine, follow commands in Chapter 5.
Appendix D

Contents of enclosed flash drive

thesis.pdf.............................. the thesis text in PDF format
lib...................................... the directory with the provided repositories
  __fire_workspace.............. the ROS package of the building simulation
  __simulation....... the ROS package that provides support for simulation
  __uav_core............... the directory with core modules for UAV
  __uav_modules .......... the directory with various ROS packages
  __README.md................... the file with repository information
  __install_ros.sh......... the bash script to install ROS on Ubuntu
src.............................. the directory of the implemented solutions
  __window_automaton......... the state machine source codes
  __window_locator... the ROS package that simulates window detection
  __window_analyser.py...... the Python script that analyses logs