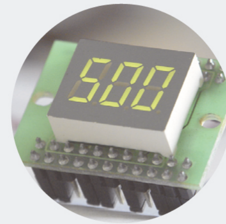
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



Coordination Control of a Robotic Lander

DAVID EMANUEL PRAZERES CARVALHO

agosto de 2022

POLITÉCNICO
DO PORTO

Politécnico do Porto

Instituto Superior de Engenharia do Porto

Coordination Control of a Robotic Lander

David Emanuel Prazeres Carvalho

Master in Electrical and Computer Engineering
Specialization Area of Autonomous Systems

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

August, 2022

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Specialization Area of Autonomous Systems.

Candidate: David Emanuel Prazeres Carvalho, No. 1160640, 1160640@isep.ipp.pt

Scientific Guidance: Eduardo Alexandre Pereira da Silva, eps@isep.ipp.pt

Scientific Co-Guidance: Alfredo Manuel Oliveira Martins, aom@isep.ipp.pt

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

August, 2022

Acknowledgements

Em primeiro lugar, agradeço aos meus orientadores, Prof. Eduardo Silva e Prof. Alfredo Martins, pela oportunidade de trabalhar nesta dissertação e pelo acompanhamento e apoio dispensado durante o mesmo.

Em especial, agradeço ao José Oliveira com quem realizei grande parte do mestrado e que teve um impacto direto nesta tese e também ao José Antunes e André Moura pelos bons momentos que passamos, por partilharem comigo o gosto da área de robótica e por me motivaram e ajudaram durante estes anos.

Agradeço ainda aos meus colegas no CRAS pelo bom convívio, boas discussões e pelo apoio que me deram durante a tese.

Por fim, agradeço aos meus pais, irmão e avó pelo apoio e motivação durante esta fase do meu percurso académico.

Abstract

Scientific and environmental focused deep sea exploration is being expanded and as such a new class of Autonomous Underwater Vehicle (AUV) capable of accessing deep underwater sea bed environment for long periods of time is being deployed. This type of vehicle and the mission environment poses challenges to the mission development as these operations contain many systems that must work together to ensure that the mission requirements are met and that the vehicle is operated safely. As such, a solution based on the SMACC library for Robot Operating System (ROS) was proposed, after the consideration between other ROS based mission execution libraries, and was tested using a simulator. The results shown were based on the simulation of three missions representative of different scenarios for a deep sea exploration AUV and they were evaluated on the completion of the mission plan and it's computational resource usage.

Keywords: AUV, coordination control, hierarchical finite state machines, mission coordination, ROS, SMACC

Resumo

Com a expansão da exploração do mar profundo com foco científico e ambiental, uma nova classe de AUV capaz de aceder ao fundo do mar por longos períodos de tempo está a ser utilizada. Este tipo de veículo e o ambiente da missão representam desafios para o desenvolvimento da missão, pois contêm muitos sistemas que necessitam de trabalhar juntos para garantir que os requisitos da missão são cumpridos e que o veículo é operado com segurança. Assim, uma solução com base na biblioteca SMACC para ROS foi proposta, após a consideração entre outras bibliotecas de execução de missão com base em ROS, e foi testada usando um simulador. Os resultados apresentados foram obtidos da simulação de três missões representativas de cenários diferentes de exploração em alto mar para um AUV e foram avaliados quanto ao cumprimento do plano de missão e ao uso de recursos computacionais.

Palavras-Chave: AUV, controlo de coordenação, maquinas de estado finitas hierárquicas, coordenação de missão, ROS, SMACC

Contents

List of Figures ix

List of Tables xi

List of Acronyms xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Objectives . . . 1
- 1.3 Thesis Structure . . . 2

2 Problem Definition 3
- 2.1 What is the problem . . . 3
- 2.2 Who has the problem . . . 4
- 2.3 Importance of solving the problem . . . 4
- 2.4 Expected Outcome . . . 5

3 Theoretical Foundation 7
- 3.1 Coordination Control . . . 7
- 3.2 Supervisory Control of Discrete-Event Systems . . . 8
- 3.3 Coordination Control of Distributed Discrete-Event Systems . . . 9

4 State of the Art 11
- 4.1 Mission Coordinator Architectures . . . 11
  - 4.1.1 Hierarchical Finite State Machines . . . 11
  - 4.1.2 Behaviour Trees . . . 12
  - 4.1.3 Petri Nets . . . 12
  - 4.1.4 Mission Coordinators Architectures Comparison . . . 12
- 4.2 Robot Level of Autonomy Metrics . . . 15
  - 4.2.1 Mobility, Acquisition and Protection (MAP) . . . 15
  - 4.2.2 Draper Three-Dimensional Intelligence Space . . . 16
  - 4.2.3 Autonomous Control Level Chart (ACL) . . . 16
  - 4.2.4 Sheridan Scale for Autonomy . . . 16
- 4.3 ROS Mission Coordinators . . . 17

4.3.1 SMACH . . . 17
4.3.2 ROS Task Manager (ROSTM) . . . 18
4.3.3 SMACC . . . 18
4.3.4 Robot State Machine (RSM) . . . 19
4.3.5 ROS Behavior Tree (ROSBT) . . . 19
4.3.6 Petri Net Plans (PNP) . . . 19
4.3.7 Comparison . . . 20

5 Coordination System **23**
5.1 System Simulation . . . 23
5.2 AUV Platform . . . 25
5.2.1 Requirements . . . 26
5.3 ROS Mission Planner . . . 26
5.4 Proposed Architecture . . . 29

6 Control and Coordination Implementation **33**
6.1 SMACC State Machine . . . 33
6.1.1 Mission State Machine File Structure . . . 33
6.1.2 Mission State Machine . . . 34
6.1.3 Mode States . . . 35
6.1.4 Super States . . . 35
6.1.5 States and Inner States . . . 35
6.1.6 Orthogonals . . . 36
6.1.7 Clients . . . 36
6.2 Developed clients . . . 37
6.2.1 Control Client . . . 37
6.2.2 Multirole Sensor based clients . . . 37
Battery Client . . . 38
DVL Client . . . 38
Odometry Tracker Client . . . 39
6.2.3 USBL Receiver Client . . . 39
6.2.4 SMACC Action Client Base based clients . . . 39
USBL Sender Client . . . 39
UUV Control Client . . . 40
UUV Control Engine Client . . . 41

7 Results **43**
7.1 Turtle Hop mission . . . 43
7.2 Two Turtle Hop Coordination mission . . . 47
7.3 Turtle Low Battery Emergency mission . . . 54
7.4 System Autonomy . . . 61

8 Conclusion **63**
8.1 Future Work . . . 64

References **65**

A Example of a mission: Turtle Hop **71**
A.1 sm_turtle_hop . . . 71
A.1.1 sm_turtle_hop.h . . . 71
A.1.2 sm_turtle_hop.cpp . . . 73
A.2 Modestates . . . 73
A.2.1 ms_turtle_hop_run_mode.h . . . 73
A.2.2 ms_turtle_hop_recovery_mode.h . . . 74
A.3 ss_hop . . . 74
A.3.1 ss_hop.h . . . 74
A.3.2 sti_hover_floor.h . . . 75
A.3.3 sti_move_to_waypoint.h . . . 77
A.3.4 sti_land.h . . . 78
A.4 States . . . 79
A.4.1 st_do_science.h . . . 79
A.4.2 st_idle.h . . . 80
A.5 Orthogonals . . . 81
A.5.1 or_navigation.h . . . 81
A.5.2 or_obstacle_perception.h . . . 82
A.5.3 or_timer.h . . . 83
A.5.4 or_usbl.h . . . 83
A.6 cl_usbl_receiver . . . 84
A.6.1 cl_usbl_receiver.h . . . 84
A.6.2 cb_usbl_receive_data.h . . . 85
A.7 cl_control . . . 86
A.7.1 cl_control.h . . . 86
A.7.2 cb_control_heave.h . . . 87
A.8 cl_uuv_control . . . 88
A.8.1 cl_uuv_control.h . . . 88
A.8.2 cl_uuv_control.cpp . . . 89
A.8.3 cb_uuv_control_check_waypoint_set_end.h . . . 89
A.8.4 cb_uuv_control_init_waypoint_set.h . . . 90
A.8.5 uuv_control_action_server.cpp . . . 91
A.8.6 cp_pose.h . . . 94
A.8.7 cp_pose.cpp . . . 95
A.8.8 cp_waypoints_navigator.h . . . 98
A.8.9 cp_waypoints_navigator.cpp . . . 99

A.9 cl_dvl 101
A.9.1 cl_dvl.h 101
A.9.2 cb_dvl_detect_floor.h 101
A.9.3 cb_dvl_detect_land.h 103
A.9.4 cp_dvl_sea_floor.h 104

List of Figures

5.1 Visualisation of the OpenFOAM solver for a Turtle 3 model Computational Fluid Dynamics (CFD). 24
5.2 Visualisation of the calculated drag in Paraview of the Turtle 3. . . . 24
5.3 Turtle 3 Autonomous Underwater Vehicle (AUV) aboard Mar Profundo. 25
5.4 State machine *sm_three_some* from *SMACC* reference library, recorded from *SMACC* Viewer. 28
5.5 Proposed architecture of the mission planner 31

6.1 Folder structure of the *SMACC* mission files. 34
6.2 Folder structure of the *SMACC* mission *include* files. 34
6.3 Folder structure of the *SMACC* mission *src* files. 35

7.1 Turtle Hop mission diagram 44
7.2 Turtle Hop mission state machine graph, recorded from *SMACC* Viewer. 45
7.3 Turtle Hop mission state machine transitions graph. 46
7.4 Turtle Hop mission position graph. 46
7.5 Turtle Hop mission resource usage graph. 47
7.6 Two Turtle Hop mission diagram. 48
7.7 Screenshot of both Turtle 3 AUVs in the Gazebo simulator 48
7.8 Screenshots of the Two Turtle Hop Coordination mission in the Gazebo simulator. 49
7.9 Two Turtle Hop Coordination mission state machine graph, recorded from *SMACC* Viewer. 50
7.10 Two Turtle Hop Coordination mission state machines transitions graphs. 51
7.11 Two Turtle Hop Coordination mission position graphs. 52
7.12 Two Turtle Hop Coordination mission resource usage graph. 53
7.13 Turtle recovery mission diagram 55
7.14 Turtle recovery mission state machine graph, recorded from *SMACC* Viewer. 56
7.15 Screenshots of the Turtle recovery mission in the Gazebo simulator and RVIZ. 57
7.16 Screenshots of the Turtle recovery mission docking in the Gazebo simulator. 58

7.17 Turtle recovery mission position graph. 59
7.18 Turtle recovery mission battery charge graph. 59
7.19 Turtle recovery mission state machine graph, recorded from *SMACC* Viewer. 60
7.20 Turtle recovery mission resource usage graph. 60

List of Tables

4.1 Highlights of Mission Coordinators Architectures 13
4.2 Comparison of Mission Coordinators Architectures 15
4.3 Comparison of ROS Mission Coordinators 20

List of Acronyms

ACL	Autonomous Control Level Chart
AI	Artificial Intelligence
AS	Autonomous System
AUV	Autonomous Underwater Vehicle
BMS	Battery Management System
BT	Behaviour Tree
CFD	Computational Fluid Dynamics
CPN	Coloured Petri Net
DDES	Distributed Discrete-Event System
DES	Discrete-Event System
DVL	Doppler Velocity Log
FSM	Finite State Machine
GPS	Global Positioning System
GUI	Graphical User Interface
HFSM	Hierarchical Finite State Machine
INS	Inertial Navigation System
MAP	Mobility, Acquisition and Protection
OOPN	Object-Oriented Petri Net
PN	Petri Net
PNP	Petri Net Plans
ROS	Robot Operating System
ROSBT	ROS Behaviour Tree

ROSTM ROS Task Manager

RSM Robot State Machine

TF Transform Library

UAV Unmanned Aerial Vehicle

UML Unified Modeling Language

USBL Ultra-short Baseline

UUV Unmanned Underwater Vehicle

VBS Variable Buoyancy System

Chapter 1

Introduction

1.1 Context

With the expansion of scientific and environmental focused deep sea exploration, a new class of Autonomous Underwater Vehicle (AUV) is being deployed, capable of accessing deep underwater sea bed environment for long periods of time.

AUVs such as the Turtle [1] mark an important step in the viability of these missions by reducing their overall cost. Traditionally, the primary cost of deep sea observation missions is the installation of stationary systems with sensors, usually requiring cables for power and communication, that use support vessels and a lot of manpower. These costs are largely reduced in an AUV based solution.

Additionally, existing classical AUV solutions are not equipped to handle the challenges found in long stays in the deep sea floor environment. However, robotic lander systems like [1] are a platform that address these problems.

Other benefits to such a solution are the ease of redeployment, again cutting the costs of the operation, and enabling different long term observation missions, such as those involving multiple locations, that would otherwise require manual relocation of the observatory.

1.2 Objectives

Within this context, the work presented in this thesis aims to accomplish the following objectives:

1. Analyse the problem from a coordination control perspective;
2. Explore existing research in literature with regard to control of autonomous vehicles;
3. Design and implement a control system for an AUV capable of performing the tasks required in the mission scenario.
4. Test the developed control system in missions representative of the requirements of deep sea bed research.

The purpose of the study of the problem is to enable a better understanding of the necessities and difficulties that exist in deep sea and sea floor missions with AUVs.

Thus, it is important to verify the existing research of similar robots with regard to their implementation of the control and coordination.

After exploring the state of the art, the option most suitable to the requirements, defined from the analysis of the problem, must be converted to a high level design and implemented.

This implementation must then be tested in suitable scenarios in order to present relevant results.

1.3 Thesis Structure

In Chapter 2, a problem definition is derived from the context of this thesis and expected outcomes from a possible solution are presented.

Next, in Chapter 3, the theoretical foundations of the work that this thesis is built upon are analysed.

In Chapter 4, the state of the art of mission coordinators, Robot Operating System (ROS) based mission coordinators and robot level autonomy metrics are overviewed.

Chapter 5 describes the planning of the project, going over the selections made and proposing an architecture for the system.

Then, in Chapter 6, the implementation of the system is explained.

The results of the simulations used for testing the system are shown in Chapter 7.

Finally, in Chapter 8, the conclusion of this thesis and future works, based on possible improvements, are presented.

Chapter 2

Problem Definition

In this Chapter, the problem is defined by answering three questions: what is the problem; who has the problem; and the importance of solving the problem. The expected outcome of a possible solution is then specified.

2.1 What is the problem

An AUV is a robotic vehicle that operates autonomously in an underwater environment. To do so, it must contain components for mobility, positioning, computation and power. In addition, depending on the operational needs, it might contain supplementary modules.

Each of the components in an AUV can be called a subsystem. In order for the vehicle to operate in a mission, each subsystem must be able to work towards the mission goal. This usually requires a specific firmware for each of the subsystem, such as in the case of the motors and sensors, as well as software that requires interaction between said subsystems, such as in the case of the positioning and mission path planning subsystems, which require sensor data and might need to interface with the motors.

Due to nature of autonomous operation and the potential harsh or hard to reach environments that these robots operate in, one major requisite is the reliability of the vehicle. A critical situation can occur when a subsystem fails or reaches a state in which an action must be taken to assure safe operation of the vehicle.

A critical situation can vary in importance and if it is anticipated or not. The priority can be mission specific, in the case of failure of the previously referenced supplementary modules, but the subsystems that permit the AUV to reach a rescue location always take precedence. Some critical situations can be expected during a mission, such as low battery charge, and have a specific actions to be taken when they happen; on the other hand, non-expected critical situations can be a much bigger risk to the vehicles integrity, as it is more difficult to prepare for those cases.

All the critical subsystems must be able to guarantee that the AUV can return to a reachable location in case of any critical situation. Therefore, all the firmware and software of the subsystems in the AUV must not fail during a mission and, if possible, have redundancy. Failure of the software can be defined as not being able to produce the expected outcome. This can be caused by the code reaching a state in which it cannot exit, or having unexpected behaviour.

As such, and to answer the question posed in this Section, it is necessary to develop a solution that allows the creation and execution of a mission for a lander type AUV that can handle the concerns previously mentioned.

2.2 Who has the problem

A mission designer's job is to develop the software in a way that guarantees that the AUV is operated safely and that all of the mission requirements are met. Due to the nature of the architecture of an AUV and all of the requirements for safe operation, this task is challenging.

The difficulty of programming a mission will vary due to it's length and complexity, which depend on how many different actions are required and how many subsystems exist in the AUV. Furthermore, with each additional action or subsystem, more points of potential failure are added to the mission. All of these make the mission design process time consuming and prone to failures.

2.3 Importance of solving the problem

Creating a solution that can simplify the control and interactions between the subsystems will lead to the viability of longer and more complex exploration and scientific missions, guaranteeing the safe operation of the hardware, and the full use of the capabilities of the robot.

As some of the missions that these vehicles perform might only differ in a few actions or subsystems, modularity is an attractive attribute to such a solution. The ability to reuse certain mission specific actions and subsystems in subsequent missions is very desirable to the overall software architecture since it helps reduce the time to create said missions. This also improves robot and mission reliability since

more time can be spent verifying the software and guaranteeing that it will behave in the expected manner.

2.4 Expected Outcome

As such, a possible solution is expected to:

1. Be able to make use of all of the robot's systems;
2. Be robust to hardware problems that might occur during the mission;
3. Be reliable in code execution, avoiding any halting during the mission;
4. Be developer friendly, in order to reduce the time of development and complexity of the design;
5. Allow easy integration of new sub-systems, without a rework of the existing software;
6. Suitable to be implemented in existing autonomous robots.

These points comprise the requisites explored in the previous Sections, and are the aim for the work developed in this thesis. It is now important to understand Coordination Control to better define concepts approached in this Chapter.

Chapter 3

Theoretical Foundation

Understanding coordination control is important in the development of a system that is capable of controlling the subsystems associated with an AUV while executing missions.

These will be explored in the subsequent sections of chapter 3.

3.1 Coordination Control

Coordination control is described by Schuppen [2] as "a form of control of a multilevel system.". It is comprised of a distributed system, a coordinator subsystem and other subsystems. The coordinator subsystem is tasked with coordinating the operation of the other subsystems in order to meet the desired control objectives of the distributed system. It achieves this by restricting the behaviour or activities of the subsystems [2]. This sort of control is required because as Schuppen [2] states, within the control theory of distributed discrete-event systems, two or more subsystems may block each other, causing the system to be unable to transition to any other state. This sort of behaviour is one of the biggest challenges in an autonomous mission since the robot must be able to perform its tasks without an operator correcting any lockups of the system.

In the context of this thesis, the coordinator is the mission control and the subsystems to be controlled belong to the robotic lander, such as the control, localisation and sensing. The main control objective is the guidance of the vehicle through the

programmed mission while maintaining a set of requirements that were considered in Sect. 2.4.

3.2 Supervisory Control of Discrete-Event Systems

As previously stated, this control system is intended to be applied in a robotic lander which, like many other robotic applications, is comprised of multiple discrete-event subsystems. These types of discrete-event systems that are composed of two or more subsystems are called distributed [3]. As these systems are very complex due to the usually large amount of components, human operators are not able to design a controller or supervisor by hand [3]. As Komenda and Masopust [3] stated "Supervisory control is a formal method of providing a theory to design supervisors for discrete-event systems.".

In supervisory control theory, automata are usually called generators. A generator is a quintuple:

$$G = (Q, \Sigma, f, q_0, Q_m)$$

where Q is a finite nonempty set of states, Σ is a finite set of events, f is a *partial transition function* where:

$$f : Q \ \times \ \Sigma \ \rightarrow \ Q, q_0 \in Q$$

$q_0 \in Q$ is the *initial state*, and $Q_m \in Q$ is a set of *marked states*. The set of all words over the event set Σ is called Σ^*. The ϵ represents an empty word, which when applied to a transition function does not perform an action[3].

The *partial transition function* f can be extended from events to words as the function:

$$\hat{f} : Q \ \times \ \Sigma^* \ \rightarrow \ Q$$

where:

$$\hat{f}(q, \epsilon) = q$$

and:

$$\hat{f}(q, aw) = \hat{f}(f(q, a)w) \text{ for } a \ \in \ \Sigma \text{ and } w \ \in \ \Sigma^*$$

Languages are used to describe the behaviour of a generator G. The language generated by G is the set:

$$L(G) = s \ \in \ \Sigma^* \mid \hat{f}(q_0, s) \in Q$$

and the marked language of G is the set:

$$L_m(G) = s \ \in \ \Sigma^* \mid \hat{f}(q_0, s) \in Q_m$$

where $L_m(G) \subseteq L(G)$[3].

A regular language is the set of behaviours of a generator. For that, a language L over an event set Σ is a set $L \subseteq \Sigma^*$ such that $L_m(G) = L$ holds true for a generator G. A language is prefix closed when $\overline{L} = L$. The prefix closure $\overline{L}$ of a language L for an event set Σ is a set of all prefixes of all its words, as in[3]:

$$\overline{L} = \{w \in \Sigma^* \mid \text{there exists } u \in \Sigma^* \text{ such that } wu \in L\}$$

A supervisor controls a system by disabling controlled events using a controlled generator. A controlled generator for a set Σ is a triple (G, Σ_c, Γ), where G is a generator for:

$$\Sigma = \Sigma_c \cup \Sigma_u$$

where Σ_c is the set of controllable events and:

$$\Sigma_u = \Sigma \setminus \Sigma_c$$

is the set of uncontrollable events, and:

$$\Gamma = \{ \gamma \subseteq \Sigma \mid \Sigma_u \subseteq \gamma \}$$

is the set of control patterns[3].

Controllability and $L_m(G)$-closedness are very important in supervisory control [4]. A controllable language is defined by a language $K \subseteq L(G)$, controllable with respect to $L(G)$ and Σ_u for a generator G over an event set Σ if[3]:

$$\overline{K}\,\Sigma_u \cap L(G) \subseteq \overline{K}$$

A $L_m(G)$-closed language is defined by a nonempty language $K \subseteq L_m(G)$ for a generator G if[3]:

$$K = \overline{K} \cap L_m(G)$$

The supervisory control problem aims to find conditions that are equivalent to the existence of a supervisor that achieves a specification. The controllability and $L_m(G)$-closedness conditions are necessary and sufficient to say that a non-blocking supervisor that achieves the specification exists [3, 4, 5].

3.3 Coordination Control of Distributed Discrete-Event Systems

The application of supervisory control of a distributed discrete-event system with synchronous communication and a global specification is a difficult problem because

it is not feasible for control synthesis to rely on the composition of all subsystems, as the complexity grows exponentially with the amount of subsystems [6]. Local control synthesis is preferred but these controllers may be blocking and, if not, may not reach the performance of the global control synthesis [6]. Hence, it is necessary to use a form of coordination between the subsystems.

In Sect. 3.2, the basic notions and concepts of discrete-event systems and supervisory control were presented. The concept of coordination control is built upon these. The first step is to identify the right parts of a global specification corresponding to each of the respective subsystems [6].

A language K is conditionally decomposable with respect to event sets Σ_1, Σ_2, Σ_k, where:

$$\Sigma_1 \cap \Sigma_2 \subseteq \Sigma_k \subseteq \Sigma_1 \cup \Sigma_2$$

if:

$$K = P_{1+k}(K) \parallel P_{2+k}(K)$$

where:

$$P_{i+k} : (\Sigma_1 \cup \Sigma_2)^* \rightarrow (\Sigma_i \cup \Sigma_k)$$

is a projection, for $i = 1, 2$[6].

For events sets $\Sigma_i, \Sigma_j, \Sigma_l \subseteq \Sigma$, the notation P_l^{i+j} is used to denote the projection from $(\Sigma_i \cup \Sigma_j)^*$ to Σ_l^*. If $\Sigma_i \cup \Sigma_j = \Sigma$ then we can simplify the notation to P_l. Additionally, $\Sigma_{i,u} = \Sigma_i \cap \Sigma_u$ is the set of locally uncontrollable events of the event set Σ_i[6].

Languages K and L are synchronously nonconflicting if $\overline{K \parallel L} = \overline{K} \parallel \overline{L}$[6].

A possible construction of a coordinator, considering two subsystems G_1 and G_2 with the event sets Σ_1 and Σ_2, respectively, and K as a specification language is as follows[6]:

1. Set $\Sigma_k = \Sigma_1 \cap \Sigma_2$ as the set of all shared events;
2. Extend Σ_k with the events $\Sigma_1 \cup \Sigma_2$ so that K and $\overline{K}$ are conditionally decomposable;
3. Set the coordinator $G_k = P_k(G_1) \parallel P_k(G_2)$ as the set of all shared events. The generator $P(G)$, for the projection, P, and generator, G, satisfies $L(P(G)) = P(L(G))$ and $L_m(P(G)) = P(L_m(G))$.

Chapter 4

State of the Art

In order to implement a mission coordinator, the architecture chosen can limit or enable what is possible to accomplish. Thus, it is an important concept to research and understand the current state of the art. The level of autonomy is also an important factor in an autonomous system and is useful to quantify a possible solution under one or more metrics. As ROS is a framework that is used in many AUV applications, identifying the available mission coordinators is essential in order to select the most appropriate one. These topics will be presented in the following sections of chapter 4.

4.1 Mission Coordinator Architectures

There are many approaches to the development of a mission coordinator for a Distributed Discrete-Event System (DDES) in a control architecture. These range from an if-statement chain, as the most basic, to complex Hierarchical Finite State Machines (HFSMs), Behaviour Trees (BTs) or Petri Nets (PNs). In the following sub-sections 4.1.1, 4.1.2, 4.1.3, the most used and relevant coordinator architectures will be discussed and in sub-section 4.1.4, they will be compared.

4.1.1 Hierarchical Finite State Machines

HFSMs are Finite State Machines (FSMs) whose states can be composed of FSMs or even previously defined HFSMs [7]. They are used in many applications in and out of robotics. In general mobile robot applications, Kurt and Ümit Özgüner present

guidelines for HFSMs system architecture and design for ground-vehicle and an interactive development methodology for additional capabilities mainly focused on the effects of new additions to existing critical behaviour[8]; Sklyarov and Skliarova describe a method for the specification of hierarchical control systems and for the design of behaviour of collaborative robots using parallelism and hierarchy with HFSMs [9]. In Unmanned Aerial Vehicle (UAV) applications, there is use of parallel HFSMs for search and rescue tasks which shows better results than classical FSM techniques [10].

HFSMs have also seen use with AUVs, primarily in gliders [11, 12, 13] and in multi-AUV formations [14, 15].

4.1.2 Behaviour Trees

Another prominent technology, that was initially developed for Artificial Intelligence (AI) in computer games[16], is BTs. They have similarities to HFSMs, their main difference being the use of tasks instead of states. It is currently a large and growing field in the robotics community. Iovino et al. have conducted a survey of BTs in robotics and AI in [16] and provide an overview of over 160 research papers. In AUV systems, Sprague et al. show advantageous use of BTs in mission-critical systems for a glider type robot [17]. Colledanchise and Natale have also covered the state-of-the-art implementation of BTs in robotics [18]. Handling concurrency and its implementation in BTs has been explored in [19].

4.1.3 Petri Nets

PNs have also seen use in control architecture applications since their introduction in 1962 [20]. In general robotics tasks, the task design, modelling and execution has been introduced in [21]; a procedure for modelling and building hierarchical and distributed control systems has been considered in [22]; a methodology for the creation of hierarchical PNs modelling the activities of multi-agent robotic systems is presented in [22]. In AUV applications, PNs have been presented in several works, with the extended Coloured Petri Net (CPN) form as a research topic in the latter works[23, 24, 25, 26, 27]. As for multi-robotic mission coordination, a methodology to model and execute coordinated missions has been presented in [28, 29, 30]; an architecture for AUV cooperation is presented in [31].

4.1.4 Mission Coordinators Architectures Comparison

In the previous sub-sections, the state-of-the-art of the most prevalent coordinator architectures for robotic missions have been overviewed. The highlights from these works have been presented in the following Table 4.1.

Table 4.1: Highlights of Mission Coordinators Architectures

	Reference	Highlights
HFSMs	[8]	The hierarchical layout of an FSM and the iterative design methodology proposed enable a certain level of design flexibility for the overall system. A point of major importance when designing a capability-based FSM is preparing for as many situations and scenarios one expects to face, and a stage-by-stage development scheme enables the addition of such contingencies.
	[9]	The results of the experiments conducted show that the proposed methods and tools make possible to construct control systems implementing hierarchical or parallel algorithms and can be used for practical applications. It was found that although hierarchical and parallel algorithms can be implemented within the same system, using hierarchical parallel algorithms is very resource consuming and many constraints have to be taken into account.
	[10]	A parallel HFSM for search and rescue tasks was presented. The partial tests showed better results when compared to classical FSM techniques, improving in overall system failure and significantly reducing computational cost.
BTs	[16]	Many problems relating to reactivity, modularity or parallel execution have been topics of research and have found solutions within the works surveyed. BTs are inherently reactive as they constantly check the conditions, switching tasks based on the perceived state of the world. This mean the size of the BT must be very large to perform tasks that involve many objects and sub tasks, making them more difficult to design manually. Some solutions to this problem are being studied, like planning algorithms or machine learning. However, they are not yet mature enough to compete with the ease of manual design, especially in cases where the BT is fairly small.

	Reference	Highlights
	[17]	Some advantages are shown in the use of a BT framework for controlling mission critical systems. It is demonstrated that the implementation can invoke safety checks with high precedence using node priority. Furthermore, with respect to versatility, it is shown that several sub tasks can be accomplished in order of priority. Lastly, with respect to robustness, it is demonstrated that controls are being chosen for priority of goal fulfilment.
PTs	[24]	This paper introduces Object-Oriented Petri Nets (OOPNs). This provides direct support for models with multiple levels of activity, and gives the modeller great flexibility in the choice of modelling components as active or passive.
	[26]	The following benefits of PNs in system modelling of AUV are described: provides well developed analysis methods which can lead to a useful detection tools within behaviour of Discrete-Event Systems (DESs); suitable tool to model or disintegrate complex systems; capability of combining multiple systems and reducing to a simple operation with no changes to its original nets; naturally oriented towards real-time modelling and analysis of DESs with no proper harmonisation-timing and where common exclusion and priorities are significant.

Both HFSMs and BTs are comparable with regards to what overall behaviour can be created, however in practice there can be significant difference[16]. This is due to their specific implementations, limiting some of the possible behaviours, or their readability and design flexibility, most important in the mission creation.

In the Table 4.2, a comparison was made between HFSMs, BTs and PNs with regards to modularity, reactivity, scalability and reusability.

Modularity is the measure of easiness to add or remove states or actions from an existing mission. BTs have a clear advantage in this regard when compared to HFSMs or PNs.

Reactivity represents the ability to create a response to an event. Both HFSMs and PNs are able to easily create immediate responses to any event, therefor being able to quickly react in emergency scenarios. On the other hand, most implementations of BTs have either more complex or less reactive solutions to events in

emergency scenarios.

Scalability is used to describe the usability of the architecture in a scenario where there are many nodes. For the most part, HFSMs are able to be implemented in large mission and still be human readable due to their hierarchical nature. In the case of BTs, there are also new solutions that help in this metric. As for PNs, they are harder to interpret when implemented in these large missions.

Lastly, reusability describes the ability to reuse parts of existing solutions to other missions. HFSMs and PNs solutions alike are usually able to transfer actions between missions. Due to their more mission-specific implementation, PNs pose a higher difficulty reusing existing actions in new missions.

Table 4.2: Comparison of Mission Coordinators Architectures

	Hierarchical Finite State Machine	Behaviour Tree	Petri Net
Modularity	-	+	-
Reactivity	+	-	+
Scalability	+	+	-
Reusibility	+	+	-

4.2 Robot Level of Autonomy Metrics

As stated in the introduction to this Chapter, measuring the level of autonomy of an autonomous system is important to quantifying its effectiveness and to be able to compare it to other solutions. In [32], Atyabi et al. identifies four main methodologies for assessing the levels of autonomy of an Autonomous System (AS).

4.2.1 Mobility, Acquisition and Protection (MAP)

Mobility, Acquisition and Protection (MAP) was developed by Hasslacher and Tilden as a way of assessing the autonomy of a robot. It uses the mobility (M_x), as the amount of dimensions the robot is able to move, M_0 representing no motion and M_5 maximum mobility; the acquisition (A_x), as the capability to extract/store/utilise energy, A_0 representing zero energy consumption and delivery and A_5 best use of planned tactics to efficiently extract/store/utilise external energy; and the protection (P_x), as the ability to protect itself against the environment, with P_0 representing defence ability and P_5 use of tools, vehicles, or materials in fight/flight tactics [33]. This metric borrows heavily from animal behaviour, using a 0 to 4 scale to symbolise basic behaviour, using 5 for lower animals and 6 for large brained animals [33], and is more focused towards military applications, which makes it less desirable for use in scientific purposes.

It mainly has limitations in its its inability to address operational characteristics of AUVs, its inability to address AUV interactions, and weak discrimination between various level of autonomy [32].

4.2.2 Draper Three-Dimensional Intelligence Space

The Draper Three-Dimensional Intelligence Space was proposed by Cleary et al. to measure the intelligence of a network of agents in a multi-vehicle collaboration setting [34]. It uses Mobility Control, ranging from none or teleoperation only to integration of multiple actions; Task Planning, ranging from none or teleoperation only to multi-agent collaboration; and Situation Awareness, ranging from none, teleoperation or sensor as conduit to integrated multi-sensor fusion, as the three metrics [34].

This autonomy evaluation is capable of addressing multi-AUV autonomous control system; however, it has limitations because of the usage of task planning as one of the metrics, since it is not necessarily an indicator of autonomy, and the usage of situational awareness, measured by the number of sensors and utilised fusion method in the system rather than identifying if the system is capable of detecting and understanding the events happening around it [34].

4.2.3 Autonomous Control Level Chart (ACL)

Clough integrated features of existing metrics and some proposed by them to assess the autonomy level of UAVs in [35], creating the Autonomous Control Level Chart (ACL). It uses a classification with levels 0 to 10 and differentiates them in Perception/Situational Awareness, Analysis/Decision Making and Communication/-Cooperation. Level 0 indicates a remotely operated vehicle; levels 1 to 2 indicates vehicles that execute one or a set of pre-planned missions; levels 3 to 5 includes vehicles with some response to faults/events; and levels 6-9 encapsulates varying degrees of multi-vehicle coordination and cooperation[35].

4.2.4 Sheridan Scale for Autonomy

Lastly we have the Sheridan Scale for Autonomy, introduced by Sheridan and Verplank, which specifies 10 levels of autonomy. It ranges from fully controlled by the operator to fully controlled by the computer [36]. It is not very relevant to the current state of autonomous vehicles, especially lander type AUVs, since all but the last level expects some sort of communication with an operator during the execution, which nowadays is usually never the case.

4.3 ROS Mission Coordinators

Developing a system compatible with ROS is important as many or all of the AUV systems will already be executing ROS based code. This also allows for easier integration of the control system with existing ROS based vehicles.

4.3.1 SMACH

The first and perhaps most used package for task and mission coordination is *SMACH*, which is a hierarchical state machine library based on python [37].

There are four provided container classes: *StateMachine*, *Concurrence*, *Sequence*, and *Iterator*. These classes define the execution semantics of groups states; the *StateMachine* container can be used to implement HFSMs state groups; the *Concurrence* container executes its states simultaneously and only transitions when all states are ready to transition; the *Sequence* container inherits from the *StateMachine* container and adds auto-generate transitions that create sequences of states from the order the states are added to the container; and the *Iterator* container which also inherits from the *StateMachine* container and adds a looping behaviour to the states to allow cycles [37].

As for states, there are five provided state classes: *State*, *SPAState*, *MonitorState*, *ConditionState*, and *SimpleActionState*. States in *SMACH* are defined by their outcomes, which is a pre-programmed property that corresponds to the outcome a task and can be connected to other states, forming transitions. The *State* class is the base state interface and it does not have any outcomes; the *SPAState* has three outcomes: succeeded, preempted, and aborted, which are three commonly used outcomes in state machines; the *MonitorState* can subscribe to ROS topics and block while a condition holds and has valid, invalid and preempted as outcomes; the *ConditionState* can execute a callback function and will outcome true or false; and finally, the *SimpleActionState* can interface with the *Actionlib* [38] and will outcome succeeded, preempted or aborted [37].

Other key features of *SMACH* are the ability to parse user data between states, which allows information like sensor data to be shared from one state to another that will execute a task based on that value; preemption propagation is also built into *SMACH* and is a very useful tool for implementing routines that can have priority over others, by being able to handle a termination signal and cancel the previous task programmatically [37]; finally, *SMACH* can be easily debugged using the provided *SMACH Viewer* [39], which includes information on the initial state labels of each container and the contents of user data structures.

4.3.2 ROS Task Manager (ROSTM)

ROS Task Manager (ROSTM) is a general task manager written in C++ that was developed with the goal of creating a simpler to use, however less expressive, task scheduler, but with the option of linking it to *SMACH* and *Actionlib*, trading off the simplicity but enabling the use of both of their strengths [40].

The main concept used in ROSTM is a task, which is a behaviour that will run for a period of time and can terminate on completion or interruption. These routines will be executed with the context of system variables, ROS topics, and services, but they can also store their own internal variables and state [40]. A sequence of these tasks can then be called in code in order to create a mission. If more advanced structures are needed, they must be combined with the integration of *SMACH*, *Actionlib* or *MoveBase* [41].

4.3.3 SMACC

SMACC is an event-driven, asynchronous, behavioural state machine library for ROS written in C++ [42]. It is built using the Boost Statechart Library [1] and inspired by Harel's statecharts [43] and *SMACH*.

One of the main features of *SMACC* is orthogonality, from Harel's work, which splits every sub-system of a robot into a container of clients, client behaviours and orthogonal components. This makes it simpler to differentiate the behaviour of a sub-system throughout the different states. In a state, an orthogonal is usually composed of one client, client behaviours are associated with that client if some task is being performed in the state, and orthogonal components are tasks that start executing from the creation of the orthogonal, independently of the state[42].

HFSMs are supported by *SMACC* but require that only the leaf states, states that do not have any state in them, have orthogonals. All other structures that contain states in them can be classified as *state machine*, *mode state* or *super state*. *State machines* are reserved for the main structure and usually there is only one per application or mission; *mode state* are main behaviour groups and usually are divided into a run mode state and a recovery mode state; super states are the simpler structure and appear any time a sequence of states is required[42].

Events may be consumed by states, which will lead to the transition to another state, or state reactors, which combine multiple events and generate a new event. Another key feature of *SMACC*, that is inherited from the Boost Statechart Library, is memory of the mode state, super state and state that an event originated from. This is extremely useful for developing recovery behaviours, as the mission can be resumed from the exact point it stopped, after the recovery sequence was executed[42].

[1] `https://www.boost.org/doc/libs/1_53_0/libs/statechart/doc/index.html`

The *SMACC* library includes some premade clients for general application as well as *MoveBase* and *Actionlib*. It also has an extensive reference library of state machines with examples of the features described. A *SMACH Viewer* like viewer is also provided in the form of the *SMACC Viewer*[42].

4.3.4 Robot State Machine (RSM)

The Robot State Machine (RSM) is a state machine application written in C++ for exploration and waypoint following that is mainly focused on inspection, rescue and similar scenarios [44]. It follows a Unified Modeling Language (UML) state pattern and offers the following basic states: *Boot State*, *Idle State*, *Teleoperation State*, *Emergency Stop State*, and *Waypoint Following State* [44]. Other than the *Boot State*, which is the initial state that performs the boot-up process, all the other states are geared towards a partially human operated vehicle.

It includes a Graphical User Interface (GUI) that allows an operator to control the state of the robot as well as set waypoints.

4.3.5 ROS Behavior Tree (ROSBT)

ROS Behaviour Tree (ROSBT) is a library that enables the use of BTs in ROS using either C++ or python [45].

It supports *Selector nodes*, which are used to find and execute the first child that does not fail, returning a status code of *success* or *running*; *Sequence nodes*, which are used to find and execute the first child that has not yet succeeded, returning *failure* or *running*; *Parallel nodes*, which return *success* if the defined minimum amount of its children return *success*, *failure* if not and *running* if not enough children have returned; *Decorator nodes*, which can alter the returned value of the child, and are implemented as *Decorator Retry*, that retries the execution if it returned *failure*, and *Decorator Negation*, that inverts the returned outcome; *Action nodes*, which perform an action and return *success* if the action is completed, *failure* if it cannot be completed, or *running* if it is still executing; and finally, *Condition nodes*, which determines if a condition has been met.

4.3.6 Petri Net Plans (PNP)

Petri Net Plans (PNP)[46] is a behaviour representation framework that enables the design of expressive plans in dynamic, partially observable and unpredictable environments [47], that is based on PNs. Although it runs independently, it includes a bridge to interface with ROS.

PNP include the following *elementary structures*, as stated in [46]:

1. *No-action* is a PNP defined by a single place and no transitions;

2. *Ordinary-action* is a PNP defined by three places and two transitions in sequence;

3. *Sensing-action* is a PNP defined by a place with multiple transitions into other places.

These PNP can be combined using the sequence, conditional, loops, concurrent execution and interrupts operators [46].

The *sequence operator* allows two PNPs to be merged and become a sequence.

The *conditional operator* uses the *Sensing-action* and three other PNPs to describe a condition.

The *loops operator*, similarly to the *conditional operator*, uses a *Sensing-action* to loop a PNPs while the sensed condition holds true.

The *concurrent execution operator* is defined by the fork and join operators. The *fork operator* is obtained by generating two token from a single one using a control transition, and the *join operator* uses the same method to generate a single token from two other.

Lastly, the *interrupts operator* handles action failures and can interrupt actions upon failure events and can then activate recovery procedures.

4.3.7 Comparison

Table 4.3: Comparison of ROS Mission Coordinators

	Reference	Benefits	Disadvantages
SMACH	[37]	Fast prototyping time; Allows the design of large and complex HFSMs; Allows introspection into the state machine, using the *SMACH viewer* GUI.	Does not support unstructured tasks; Does not work well with low-level systems as it is designed to be a task-level architecture; Python is usually slower in execution when compared to similar compiled options.

	Reference	Benefits	Disadvantages
ROSTM	[40]	Very simple to use; Can link the task scheduler with *Actionlib* or *SMACH*.	Lacks functionalities unless using the linked *Actionlib* or *SMACH* functions and classes; Not easy to implement more complex tasks that are supported by other coordinators.
SMACC	[42]	Uses the Boost Statechart Library, inheriting compile time validation checking. Orthogonals make adding and utilising the subsystems in a robot very clear and easy; Can very effectively implement recovery modes and keep track of the previous states; Includes a reference library with many example state machines and clients; Like *SMACH*, it includes a visualisation tool, the *SMACC Viewer*.	Lack of documentation besides examples and doxygen increase the learning curve.
RSM	[44]	Includes a GUI for control the states by an operator.	Is very focused on human operation of the mission; Only tested in 2D applications with ground vehicles.
ROSBT	[45]	Has a GUI for the creation of a tree; Compatible with both C++ and Python.	Only has basic BTs functionality.
PNP	[46, 47]	Powerful tool for the creation of behaviours, especially in multi-robot applications.	Interfaces with ROS using a bridge instead of being a native package.

Chapter 5

Coordination System

In this chapter, the choices made for the mission planner, simulation environment and AUV platform, in relation to the development of the project, as well as the proposed architecture are presented.

5.1 System Simulation

There was a necessity to create a simulator for a new version of the Turtle lander in order to able to test and validate new software more readily and in a safe environment. Inspired by the previous work [48], one was developed using Gazebo [1] and the Unmanned Underwater Vehicle (UUV) simulator [2] [49].

This simulator uses Fossen's equations of motion [50], lift, drag and sensor plugins, 3d current velocity models simulation, and a thruster module that converts angular velocities to thrust force output. All these features make it a well-suited simulator for use with the developed mission control system.

The drag component required for the linear and quadratic damping coefficients for Fossen's equation of motion were calculated with Computational Fluid Dynamics (CFD) using OpenFOAM [3], shown in Figure 5.1, and processed in Paraview [4] based on the 3D model of the vehicle exposed in Section 5.2, with the results for one simulation shown in Figure 5.2.

[1] `http://gazebosim.org/`
[2] `https://uuvsimulator.github.io/`
[3] `https://www.openfoam.com/`
[4] `https://www.paraview.org/`

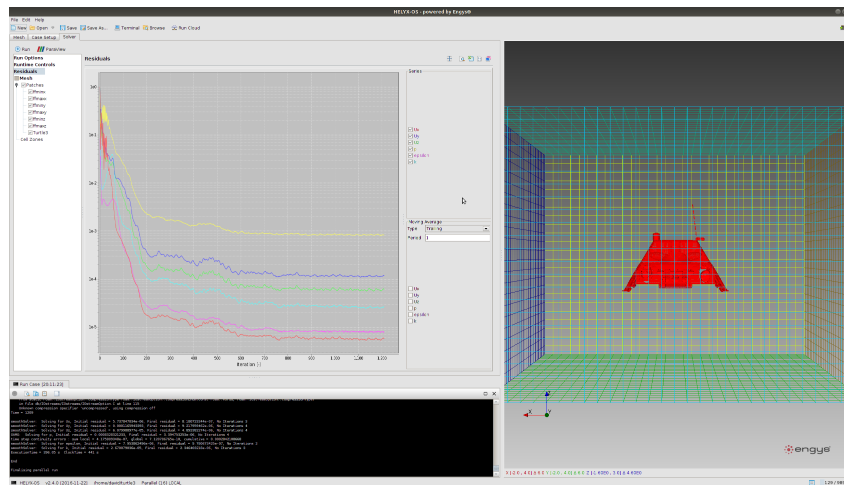


Figure 5.1: Visualisation of the OpenFOAM solver for a Turtle 3 model CFD.

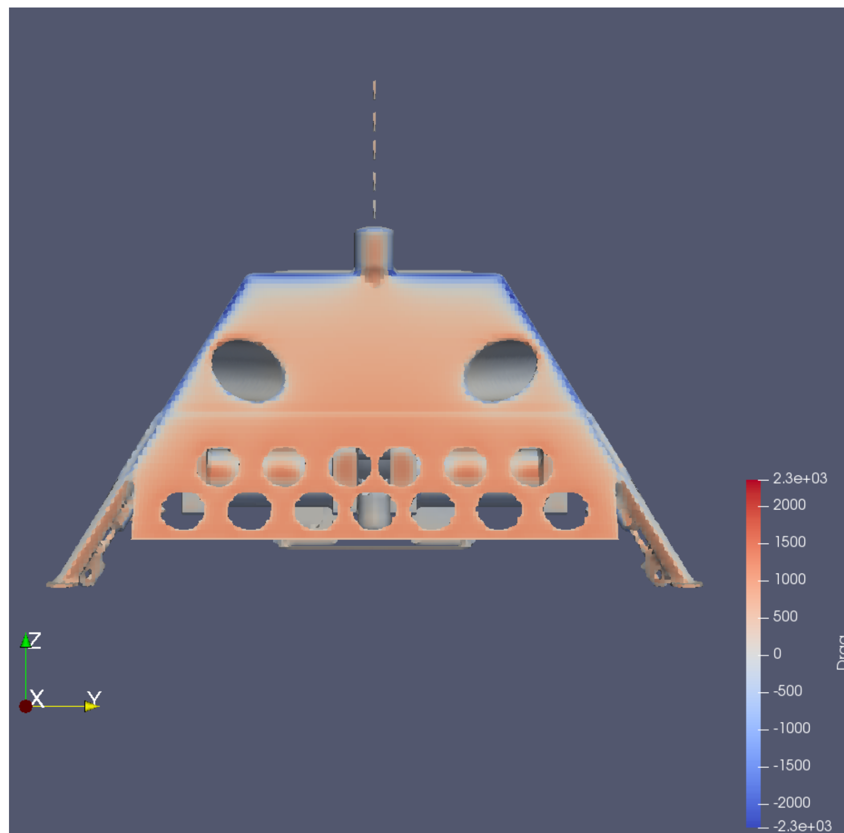


Figure 5.2: Visualisation of the calculated drag in Paraview from processed data taken from an OpenFoam CFD applied to the Turtle 3 model.

As for a state estimation and positioning system, the *robot_localization* ROS package was used. This package allows the fusion of the positional data generated by all the sensors present in the AUV and also takes into consideration the uncertainty

in the form of covariance.

5.2 AUV Platform

The platform chosen to be the aim of an initial implementation of the proposed mission control system was the Turtle 3 AUV lander, as seen in Figure 5.3. This robot is the third iteration of this lander type AUV and its selection was based on the existence of the simulator described in the previous Section, as well as the possibility of field experimentation in a real scenario.

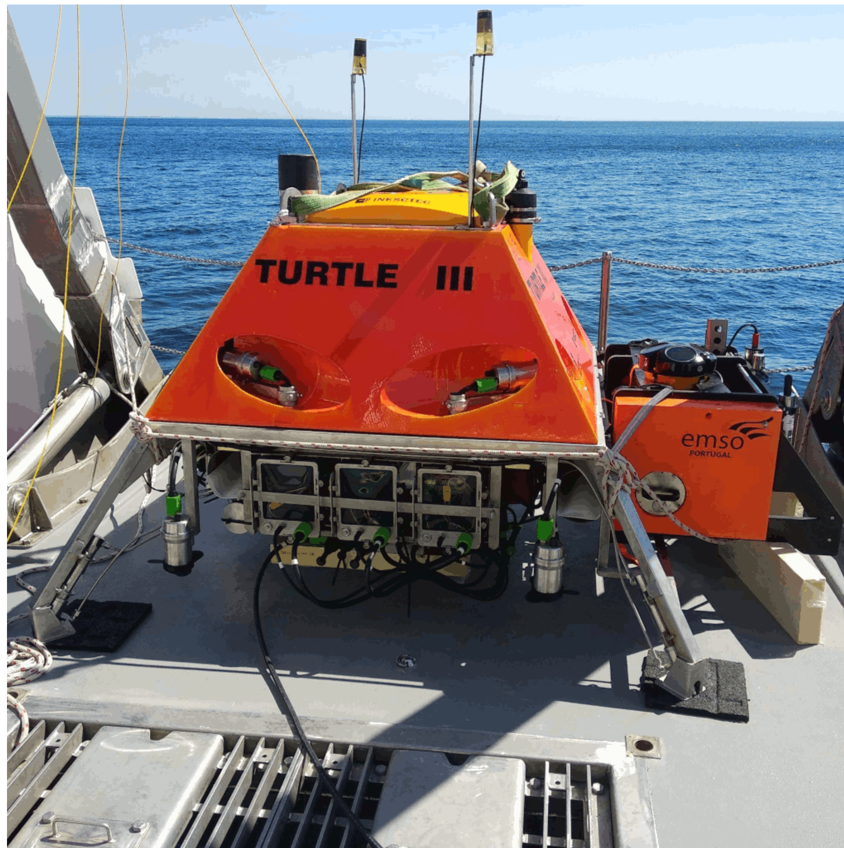


Figure 5.3: Turtle 3 AUV aboard Mar Profundo.

This new AUV version is smaller than its predecessor; capable of reaching up to a depth of 4000 m; is equipped with larger batteries allowing for longer missions; includes 8 thrusters that allow control in 5 axes and a Variable Buoyancy System (VBS); handles its navigation with an Inertial Navigation System (INS), pressure sensor and Doppler Velocity Log (DVL); uses a Ultra-short Baseline (USBL) for underwater communication and acoustic positioning when within range; and when surfaced it has a Global Positioning System (GPS) receiver for position fixes. All these subsystems in use are implemented and controlled using existing ROS packages.

These vehicles can also be outfitted with extra sensors or payloads depending on the specific needs of the mission. To allow for more complex and specific missions, these systems must be able to interact with the AUV, to be activated at the correct time and location. In some specific missions and during prototyping, manual control by a human operator at any point of the mission is also a desirable capability. The ability to respond to problems such as low battery power or mechanical failure is important for this class of vehicles, which are usually deployed at extreme depths, making recovery missions very difficult and costly.

5.2.1 Requirements

With the expected outcomes described in Section 2.4 in mind, the following requirements were selected in order to solve the problems posed in Chapter 2:

1. The mission control system must be able to make use of all of the robot's systems;
2. The mission control system should allow easy integration of new sub-systems, without a rework of the system.
3. The mission system must be able to respond to errors and cannot get stuck in a state it was not programmed to;
4. The mission design should be operator friendly, using combinations of base tasks to build a mission, therefor reducing the complexity of the design effort;
5. The mission design should allow change, like addition or reduction of tasks, without requiring major rewriting of the already created mission;
6. The mission control system must be compatible with existing code in the ROS framework.

The requirements listed are a base guideline for a control system that would not impose problems that are generally universal in the development for AUVs. More specific problems to the lander class of AUVs haven't been established due lack of maturity in this area.

It is important to find a solution that works with ROS as it is a widely used framework for AUVs and it has a big catalogue of code packages that are used throughout every sub-system that may be included in the robot.

5.3 ROS Mission Planner

The coordinated control of all of the subsystems and ability to respond to externally triggered events is an important factor for a lander type AUVs, such as the Turtle

3 described in Section 5.2. The requirements outlined in Section 5.2.1 are also necessary capabilities of the developed mission control system.

As such, after the consideration of the multiple options available for ROS, researched in Section 4.3, *SMACC* was chosen.

SMACC allows the development of interfaces for the specific subsystems as orthogonals, which can then be used in any mission or even vehicle. This reduces the work for following missions, as all the common subsystems can be used without extra implementation time.

Orthogonals in *SMACC* can be used to divide a robot into parts. They serve as a container for clients, client behaviours or client components. An orthogonal can host one or more client inside of them and can be thought of as a namespace for a client. This is useful to be able to differentiate between multiple instances of the same client in a state machine. All orthogonals defined in the mission are created at startup of the state machine and are inherited by every leaf state in that state machine. This means that the clients and client components that are associated to the orthogonal are always executing from the start of the mission onward. When a state creates a client behaviour, it will exist within the orthogonal for the duration of that state.

Another important capability is the memory of the state that an event originated, which enables recovery procedures to return to the state the robot was in before their calling. This can also be used with a manual operation.

Core *SMACC* concepts can be translated into coordination control theory. With the use of a reference state machine provided by the *SMACC* library, shown in Figure 5.4, an analysis will be made in order to provide an example of the underlying coordination control concepts in this *SMACC* application.

The state machine *sm_three_some* is a base example that includes most of *SMACC*'s functionality. It is composed of a modestate, *MsRun*, four leaf states, *StState1*, *StState2*, *StState3* and *StState4*, and two superstates, *Ss1* and *Ss2*, composed of the inner states, *StiState1*, *StiState2* and *StiState3*. Additionally, four orthogonals exist in this state machine, each with one client.

As stated in Section 3.2, a generator is composed of states, events, a transition function, initial state and marked states. As we are dealing with a HFSM, we can decompose each modestate and superstate into a generator, making the analysis process easier to interpret.

The first generator we consider is the modestate *MsRun*. It is composed of the state set: *StState1*, *StState2*, *StState3*, *StState4*, *Ss1* and *Ss2*; the event set is composed of: *EvTimer*, *EvKeyPressN*, *EvKeyPressP*, *EvKeyPressA*, *EvKeyPressB*, *EvKeyPressC*, *EvAllGo* and *EvLoopEnd*; the transition function is represented graphically in Figure5.4; the initial state is *StState1*; the marked state is *StState4*.

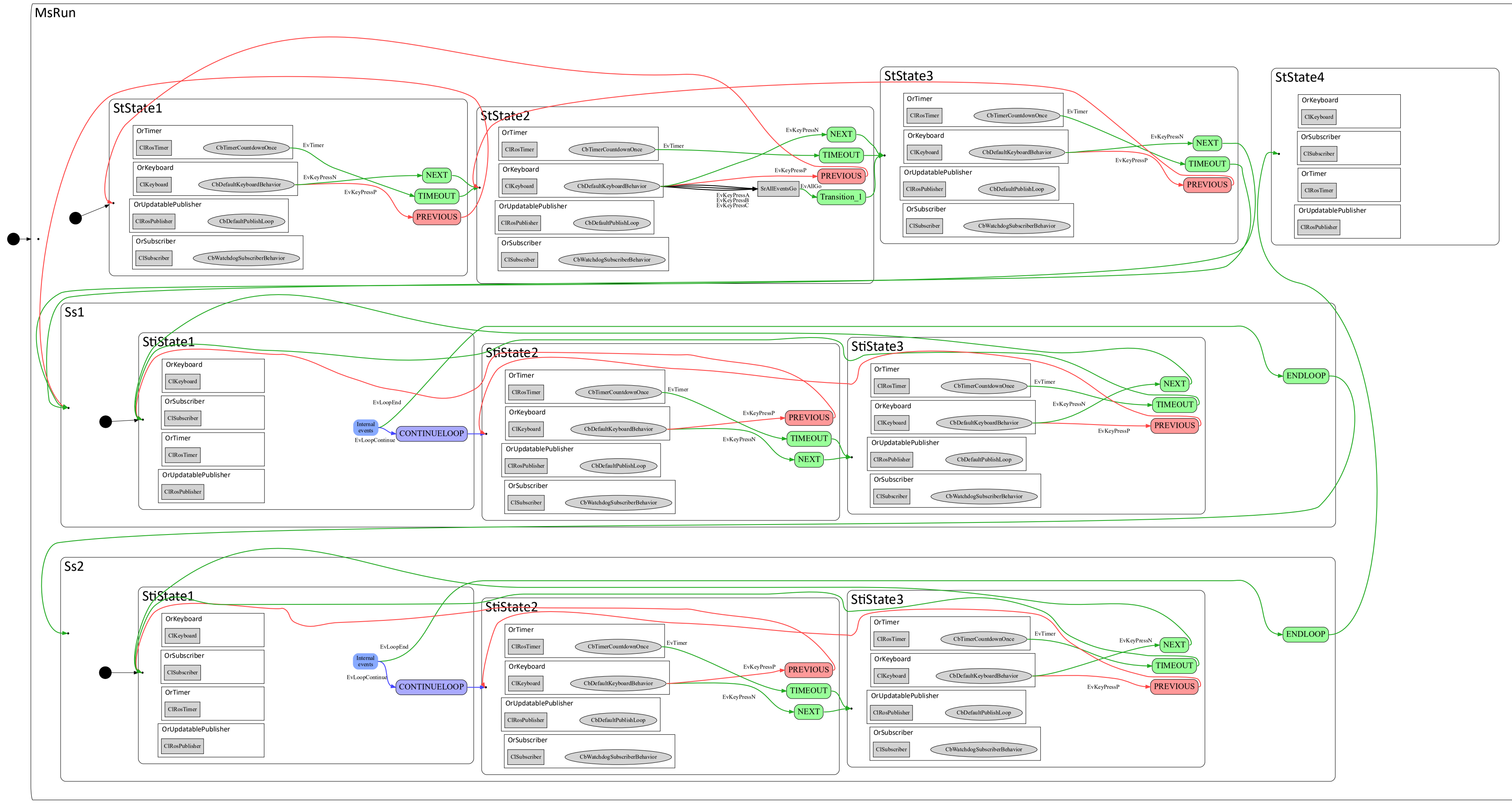


Figure 5.4: State machine *sm_three_some* from *SMACC* reference library, recorded from *SMACC* Viewer.

Both superstates in this example are equal, so only one will be analysed. They include the state set: *StiState1*, *StiState2* and *StiState3*; the event set: *EvLoopEnd*, *EvLoopContinue*, *EvTimer*, *EvKeyPressN* and *EvKeyPressP*; the transition function is again represented graphically; the initial state is *StiState1*; there is no marked state.

The *sm_three_some* is already a controlled generator. We can see examples of this control in *StState2* where the state reactor *StAllEventsGo* only triggers the event *EvAllGo* if the events *EvKeyPressA*, *EvKeyPressB*, *EvKeyPressC* have been seen. This is also the case inside the superstates, where an internal decrementing variable decides the triggered event of the initial state.

Inside each client and client behaviour on the orthogonals, more generators are running for each of the sub-systems. A similar analysis could be done for each of these but, in practical applications, there is an abstraction layer between the automata and the mission design in *SMACC* and this evaluation of the state machine is not done.

5.4 Proposed Architecture

The architecture proposed in Figure 5.5 was developed taking into consideration the existing hardware and software for the chosen AUV platform, the usage of *SMACC* as the task manager and the initial outlined requirements.

There are five initial clients associated with the orthogonals that provide actions for the mission states by interfacing with the existing robot ROS nodes. These are the *Mission Interface Client*, the *Obstacle Perception Client*, the *Navigation Client*, the *Battery Stack Client*, and the *Remote Operation Client*. Clients can be created and associated with an orthogonal for any number of subsystems that exist in the robot and are required for the mission.

Starting with the *Mission Interface*, this client interfaces with the communication nodes of the robot, allowing external commands to be sent to the robot during the mission, or for the robot to send messages or data. This client publishes and subscribes to two different ROS topics to allow easy integration. Any communication device that is associated with the AUV, e.g., the on-board USBL, is only required to publish the data it receives and send the data subscribed to the same topics the client uses. The data received can be parsed and generate events that can start, halt, or stop the mission or even switch the mode state to teleoperation.

The *Obstacle Perception Client* uses the information received from the DVL sensor to detect the depth to the seafloor. This is an important measurement for a lander AUV, whose main objective is the safe landing on the bottom of the ocean. It is composed of a client component, that updates an internal variable with the most recent sensor information about the seafloor distance, and a client behaviour,

for interpretation of the sensor data and event generation. It is a basic client that can be used in combination with the navigation to descend to a specific depth from the bottom or land. It does so by generating an event when the seafloor is detected, containing the measured distance, which can then be used by the mission navigation.

All of the robot's autonomous movements are handled by the *Navigation Client*. It includes a *Waypoint Navigator* client component, which handles the execution of pre-generated sequences of waypoints; the *Pose* client component, that updates an internal variable for the AUV's pose from the latest Transform Library (TF) transform; and a *Local Planner* client component that is associated with an action server, which handles movement requests from the client behaviours, and executes them. The *Local Planner*'s action server allows for tasks to be preempted and returns success if the goal is reached. Associated with this client are also behaviours for relative movement with the current position, surge, heave, sway and yaw rotation, global position movement as a go-to point, and execution from a list of predetermined waypoints. All of these behaviours can generate an event related to the returned status of the task. The relative and global movements use the pose from the *Pose* component to generate the goal, which is then sent to the planner. The waypoint list following behaviour interacts with the *Waypoint Navigator*, making it send a waypoint to the planner and incrementing the list index. The *Local Planner* receives goal requests and converts them to a path which is then published to the ROS *control* node of the robot.

Next, we have the *Battery Stack Client*, who much like the *Obstacle Perception Client*, is composed of a client component that updates the battery information internal variable, and a client behaviour that processes said variable. However, the case of the battery state is mission-critical, since running out of energy would most likely cause the lander to descend and become stuck in the seafloor. As such, the client behaviour can be used to monitor the information received from the battery's Battery Management System (BMS), and generate an event that transitions to a recovery or emergency mode, which can then handle the situation accordingly.

Lastly, the *Remote Operation Client* is used in a teleoperation mode when it is desired to control the AUV remotely using a joystick. The client behaviour sends twist messages, instead of paths, to the *control* node, subscribed from the *teleop* node that is usually used in ROS.

Aside from the clients and orthogonal, a mission is also composed of mode states. It is required to have at least one mode state, the *Run* mode, which contains the main mission state machine. In our architecture, there are, however, two other mode states; the *Recovery* mode handles errors or problems that might occur; the *Teleop* mode allows manual operation of the robot. The transition between mode states can be tracked, which permit a mission to return to its original state after a recovery or teleoperation mode.

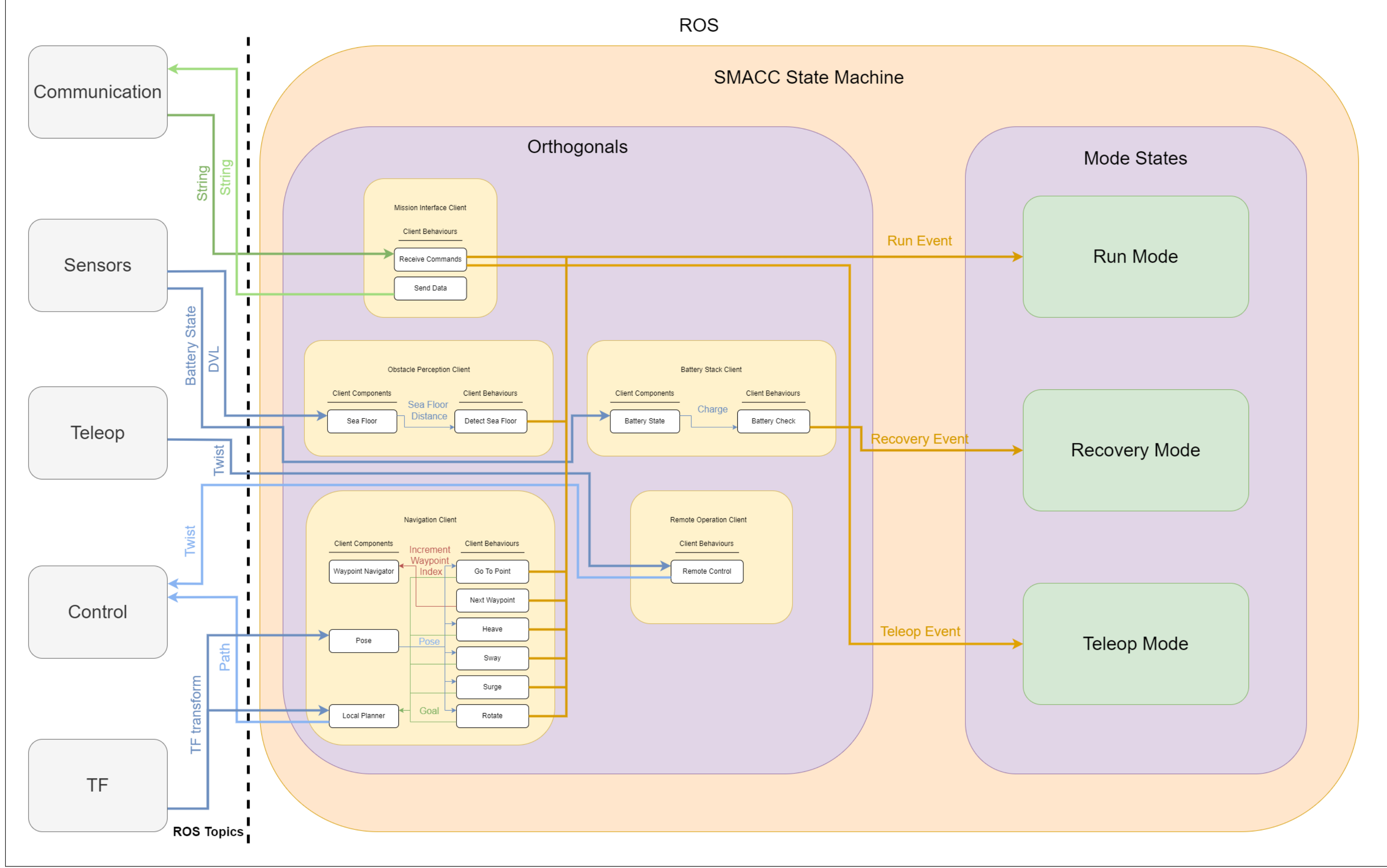


Figure 5.5: Proposed architecture of the mission planner

Chapter 6

Control and Coordination Implementation

In this Chapter, the implementation of a mission in *SMACC* will be described, as well as the development of the interfaces for the clients described in the previous Chapter 5.

6.1 SMACC State Machine

SMACC mission's have a structure to the files used and class definitions. These will be explored in the following Sections.

6.1.1 Mission State Machine File Structure

The file structure of a mission in *SMACC*, represented in Fig.6.1 is equal to any other ROS package. However, the *include* and *src* folders have a specific arrangement.

Inside the *include* folder, shown in Fig.6.2, there are folders for the clients, modestates, orthogonals and states, which is where they are defined. There is also the mission header file referenced in Section 6.1.2.

The *src* folder structure, shown in Fig.6.3, contains the folder for clients' code, which is used when used clients aren't included from outside libraries, and the source mission code file that includes the mission header file.

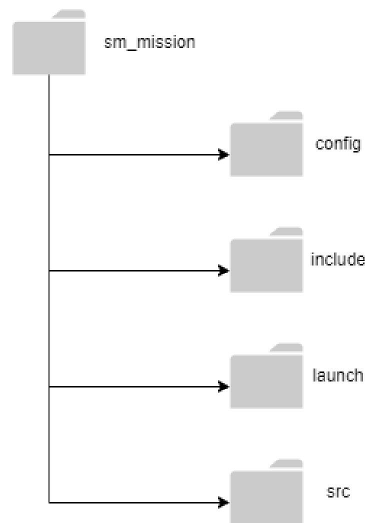


Figure 6.1: Folder structure of the *SMACC* mission files.

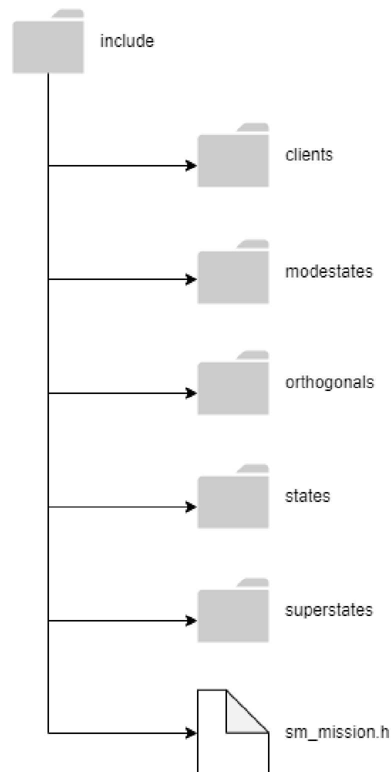


Figure 6.2: Folder structure of the *SMACC* mission *include* files.

6.1.2 Mission State Machine

In *SMACC*, the mission header file is where all the client behaviours, orthogonals, states, super states and mode states are included. The code base uses namespaces for structuring externally included code libraries like *smacc* or *client classes*, the super state classes and the mission class.

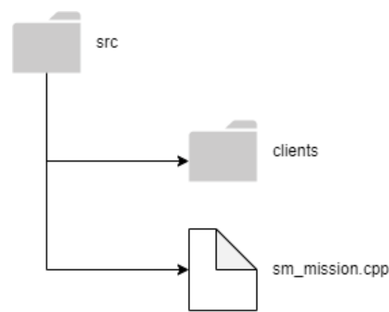


Figure 6.3: Folder structure of the *SMACC* mission *src* files.

On the initialisation of the state machine class, the state transition history variables and the orthogonals that are used are created. Additionally, global variables may also be included in this section.

The cpp file that includes this header file is where the main function is called. In there, the ROS initialisation function is called, creating a ROS node, and the mission class object is created.

6.1.3 Mode States

Inside this folder, there are the *run*, *recovery* and *teleop* modes header files.

The mode states are only defined in the include folder structure. This is because the class source code that they inherit is included from the *SMACC* library. The mode state class declaration inherits from the *SMACC* class *SmaccState* and is linked with itself, the mission state machine class, and the starting state.

Inside of the mode state class, the state transitions must be declared. This is done by specifying the event that triggers the transition and the target state. The orthogonal associated with the transition may also be included for better visualisation in the *SMACC Viewer* GUI.

6.1.4 Super States

Super states are defined in their own folder space. They are associated with inner states, which must be forward declared in the super state header file. The state declaration is similar to a mode state, requiring the linkage to itself, the mode state it is associated to, and its starting inner state.

6.1.5 States and Inner States

States are declared similarly to mode states, with the main difference being the linkage to only itself and a mode state or inner state, instead of the state machine class and to a starting state. The distinction in the association results in either a state or an inner state, respectively.

Every state has a *staticConfigure* function, for static configuration of the orthogonals; a *runtimeConfigure* function, for run-time configuration of the orthogonals; a *onEntry* function, that executes after the *staticConfigure* and *runtimeConfigure* functions. an *update* function, that requires the inheritance of the *ISmaccUpdatable* class, and is executed at a defined frequency; and an *onExit* function that is executed on the exit of the state.

The client behaviours that are executed in a state are defined in the *staticConfigure* function, with the *configure_orthogonal* function. Some states might depend on run-time information, such as data from a sensor, in which case, the *runtimeConfigure* function may be used to insert the information into a client or client behaviour defined in the static configuration.

6.1.6 Orthogonals

Each orthogonal has an associated header file. In it, the used client and component header files are included and the orthogonal class is declared. The declaration of the class is done inside the state machine namespace.

The orthogonals used by the mission state machine are initialised using the *onInitialize* function, declared in the header file. Inside, the clients and components associated with the orthogonal are created, using the *createClient* or *createComponent*, respectively.

6.1.7 Clients

Clients are the tool used by *SMACC* to perform tasks inside the state machine. The *SMACC* library offers four base client classes to be inherited by a client.

The *smacc_action_client* base class contains action servers functions, and is used when an action server client is required.

When a client is required to interface directly with either a ROS topic or service, there are three options in base clients. The *smacc_publisher_client* includes functions for publishing to a ROS topic and for subscription, the *smacc_subscriber_client* can be used. Finally, the *smacc_service_client* can create calls to ROS services.

Additionally, more clients are also available in the *smacc_client_library*, which can be directly used if possible or have useful functions to be inherited by custom clients.

All the client bases described have the *SMACC* event system integrated, allowing for state change depending on the feedback or result of the tasks being performed.

In the declaration of a client, the relevant parameters will be initialised, e.g., the ROS topic name for a subscriber or publisher. This declaration is done in the client header file, inside the mission state machine and client namespace.

To perform a task inside a state, a client behaviour must be called. These behaviours are associated with a client. Inside a client behaviour, the events that will be used are declared; the *onEntry* function that handles the initialisation of the task is defined; and the remaining required code for the task execution is also included.

A task that is intended to cause state change will use the *postEvent* function with one of the declared events to interface with the state machine.

A client may also have components, which unlike client behaviours, are started in the orthogonal initialisation, instead of a state. They inherit the *ISmaccComponent* class but may also inherited other class behaviour classes. The header and source files are usually stored in their own folder inside the client folders in the *include* and *src* folders.

6.2 Developed clients

Several clients where created for interfacing with the AUV's sub-systems in the simulation. These are implemented with available clients in the *SMACC* library and will be explained in the following Subsections.

6.2.1 Control Client

The control client was developed using the *SmaccPublisherClient* as a base. This client publishes to the *"cmd_vel"* topic, which allows direct speed requests to the motors.

One client behaviour was developed for this client, the *cb_control_heave*. As the naming of the client behaviour implies, it allows control of the heave speed of the robot. Other behaviours, such as surge, sway or rotations can also be easily implemented as needed.

This client behaviour uses the *ISmaccUpdatable* class to provide an update function, which triggers at a user defined interval. Using the *SmaccPublisherClient* provided *setMessage* function, the ROS "*geometry_msgs/Twist*" message is defined with the speed value passed into the behaviour initialisation function. The update function then calls a provided publish function which publishes the message to the topic.

6.2.2 Multirole Sensor based clients

The clients in this Section are all implemented using the *ClMultiroleSensor SMACC* class. This class extends the functionality provided by the *SmaccSubscriberClient*

class by including a timeout option associated with the subscribed ROS topic messages. It does so by creating a timeout message event when the user specified timeout period is reached.

Battery Client

In order to interface with the simulated battery associated with the AUV, the battery client was developed. This client subscribes to a topic that publishes a *"sensor_msgs/BatteryState"* message.

Two client behaviours were created for this client, the *cb_check_battery_level* and the *cb_charge_battery*, using the *CbDefaultMultiRoleSensorBehavior* class. The former behaviour uses the *onMessageCallback* function, which is called whenever a new message is published to the topic, to compare the received battery percentage to a set percentage and, when inferior, post a low battery event. The latter behaviour communicates with the simulated battery to start simulating charging, by publishing a message to a specific topic associated with the battery simulation node, and monitors the received messages, with the *onMessageCallback* function, to check if the percentage is superior to a set value and posting a battery charged event when it is so.

DVL Client

For some of the missions, it was necessary to interact directly with the DVL sensor topic, as the distance to the sea floor was necessary for path planning. The DVL client subscribes to a topic that publishes a *"uuv_sensor_ros _plugins_msgs/DVL"* message.

This client has two client behaviours that were created using the *CbDefaultMultiRoleSensorBehavior* class. The first behaviour, *cb_dvl_detect_floor*, uses the *onMessageCallback* function to post a sea floor detect event when the distance measured by the DVL is under a user specified target height. The second behaviour, *cb_dvl_detect_land*, uses the same callback function, associated with to received messages from the topic, to check if the distance to the sea floor is under the minimal range the DVL can measure, for a set duration. When it is so, a land detect event is posted.

To help with the landing detect behaviour, the *cp_dvl_sea_floor* component was also created for this client. It uses the *ISmaccComponent* class and implements a simple ROS subscriber to the DVL topic. This component reads each DVL message and stores it, when it has a valid, non negative, reading, in a private variable available through the *getSeaFloorDistance* function. This value is useful since an invalid DVL distance can be produced from the target being too far away or too close, and

by checking the previous valid measurement, one can distinguish between the two scenarios.

Odometry Tracker Client

The odometry tracker client was developed for the simulation state machines and its objective is to subscribe to the odometry ROS topic. This client subscribes to a *"nav_msgs/Odometry"* ROS message.

The *onMessageCallback* function is used in the *cb_abs_speed_track* client behaviour, to calculate the current speed of the vehicle. It generates an event when the target speed, specified in the initialisation of the client behaviour, is equal to or inferior to the actual speed. This behaviour is used in some missions to check if the vehicle has slowed down past a specified speed.

6.2.3 USBL Receiver Client

The USBL receiver client inherits the *ISmaccClient* class and is used to interface with the messages received from the USBL. The *ISmaccClient* is the base client class and it doesn't provide any additional functions.

There is a single client behaviour associated with this client, the *cb_usbl _receive_data*, which inherits the *ISmaccClientBehavior* and *ISmaccUpdatable* classes. Inside the *ISmaccUpdatable*'s *update* function, a flag is checked, inside a component that interprets the messages, and an event relative to that flag is posted if the flag was set.

This client also includes the component *cp_usbl_receiver*, referred to in the previous paragraph, that subscribes to the topic that publishes *ros_usbl _sensor/USBLData*, containing the data that was received in a specific transmission. It compares the received data to specific commands and sets a flag when it is equal. These private flags are each accessed and modified by a pair of *getFlag* and *resetFlag* functions, which return and reset a flag, respectively.

6.2.4 SMACC Action Client Base based clients

The clients in this Section are all implemented using the *SmaccActionClientBase SMACC* class. This class allows the interaction with the ROS package *Actionlib* by providing the *sendGoal* and *cancelGoal* functions and by automatically generating the *onSucceeded*, *onAborted*, *onPreempted* and *onRejected* events depending on the action server's response.

USBL Sender Client

The USBL Sender Client was implemented as an interface to the transmission of signal through the USBL sensor. This client interacts with the action server using

USBLSender actions. These actions are comprised of the *ros_usbl _sensor/USBLCommand* message, a state and a result.

Three client behaviours were developed: *cb_usbl _sender_ping*, *cb_usbl_sender _ping_loop* and *cb_usbl _sender_transmit*, which allow the AUV to send a single ping, a ping periodical or a message through the USBL sensor, respectively. The first behaviour defines the goal variable inside the *onEntry* function, which is only executed once, and then sends it using the *sendGoal* function. The second behaviour inherits the *ISmaccUpdatable* class and uses it's *update* function to define the goal variable and send it through the *sendGoal* function. The final behaviour, similarly to the first one, uses the *onEntry* function to create the goal, differing only in the data that is sent.

The *USBLSenderActionServer* is the action server that interacts directly with the USBL topics. It receives the data sent by the clients and passes it to the specific topic. If the action server sends the data, it will trigger the *onSucceeded* event; if it receives another request before it has sent the data, it will not send it and trigger the *onPreempted* event.

UUV Control Client

The UUV Control Client was created in order to interface with the waypoint following system implemented in the UUV ROS package. This client interacts with an action server using *UUVControlInitWaypointSet* actions. These actions are comprised of a vector of *uuv_control_msgs/Waypoint* messages, a state and a result.

There are two behaviours available to this client, *cb_uuv_control_init_waypoint _set* and *cb_uuv_control_check_waypoint_set_end*. The former uses the *onEntry* function to create and send a goal with the list of waypoints to follow. The latter is used, when the waypoints are being read from a file, to check if the waypoint index is at the end of the list; on the *onEntry* function, a component is used to check if the waypoint index is at the end of the list, posting a waypoint set end event if the returned value is true.

The *uuv_control_action_server* action server is used to convert a waypoint or waypoint list, received in the goal, to a format accepted by the ROS service, provided by UUV, that creates a trajectory for the AUV to follow. When a goal is received, it is converted to the correct format and sent to the *InitWaypointSet* service. While the UUV control is following a trajectory, it is publishing it's positive state to a topic. This state is monitored by the action server; when it is no longer tracking a trajectory, the server posts a *onSucceeded* event; in the event that a new trajectory is sent during the tracking, the previous trajectory will stop and the server will post a *onPreempted* event.

Along side the behaviour and action servers, two components were also created. The first behaviour, *cp_pose*, updates a private pose variable with the TF

transform of the AUV. This variable can then be accessed with the *toPoseMsg* function or *toPoseStampedMsg* function if there is a need for a timestamp of the pose. The second behaviour, *cp_waypoints_navigator*, allows the initialisation of a list of waypoints from a YAML file, at the start of a mission. It provides a *getNextWaypoints* function, which returns a waypoint and increments the waypoint index, and a *waypointSetComplete* function, which checks if the last index of the list has been reached.

UUV Control Engine Client

The UUV Control Engine Client allows the interaction with the working state of the motors on the UUV, allowing the toggle between the on and off states. This client interacts with an action server using *UUVControlEngine* actions. These actions are comprised of a of command byte, with each bit controlling the state of a different thruster, a state and a result.

A single behaviour was implemented, with the name *cb_control_engines*, which sends a goal to the action server, from the *onEntry* function.

The *uuv_control_engine_action_server* is the action server that interfaces with the UUV engine control ROS service, *SetThrusterState*. The server checks each individual bit of the command variable inside the goal and sends the corresponding state to each of the eight services, one for each thruster. This server then enters a loop where it keeps reporting the feedback of action, which is the command sent in the goal. It will then only exit the loop if another goal is sent to the server, after which a *onPreempted* event will be posted.

Chapter 7

Results

In an effort to validate the implementation of the proposed solution, three missions were designed. These cover a broad range of settings, focused on applications relevant to the Turtle 3 AUV. In the following sections, each mission will be described and the gathered results will be presented.

7.1 Turtle Hop mission

The Turtle hop mission was designed to represent the most common lander AUV mission type.

The mission description is as follows:

1. Start from the surface of the sea, after a start command;
2. Move to a waypoint on the sea floor, read from a previously configured list;
3. Wait some time to simulate collecting data from sensors;
4. Repeat points two and three until all waypoints have been reached.

In Figure 7.1, the mission is shown as it was formerly described, with a numerical description for the expected sequence of actions and representative trajectory between waypoints.

The following state machine in Figure 7.2 was implemented to execute the described mission.

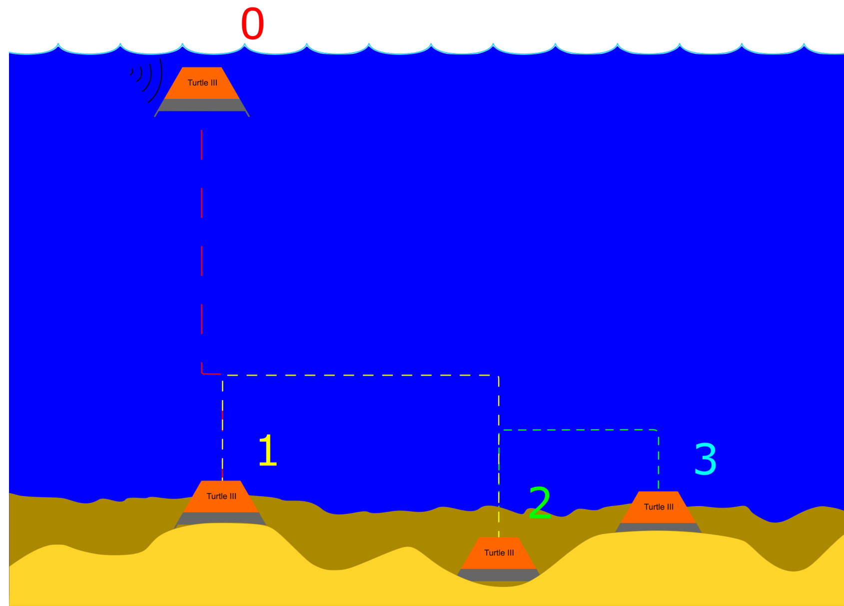


Figure 7.1: Turtle Hop mission diagram

The Figure 7.3 shows the transitions between the states over the mission duration, where the bars can represent modestates, superstates, states or inner states and the dotted lines at the end of each bar represent the event that triggers the state transition. It is possible to see that all the states run inside the *MsTurtleHopRunMode* modestate. The initial state is *StIdle* which waits for a start signal to be sent to the USBL sensor. The *SsHop* super state contains all necessary states to move between waypoints.

The waypoints configured in this mission are the following: (10, -5), (-5, -5), (-5, 10) and (10, 10), where the first value indicates the X coordinate and the second the Y coordinate. These waypoints were configured in the local reference frame of the robot, where (0, 0) is the starting position. The path chosen to move the AUV between waypoints, as can be seen in the position graph in Figure 7.4, dives or ascend until twenty meters above the sea floor, then moves in the XY plane until above the target waypoint and finally descends and lands. This was done to avoid potential collisions while moving laterally with higher terrain features in the sea floor. The *StDoScience* state is transitioned to when the robot is landed at the waypoint; it only contains a thirty second timer to simulate a scaled down period, where the AUV would be performing it's assigned mission task. When all four waypoints on the list are reached, the state machine was configured to transition back to the *StIdle* state.

In the Figure 7.5, the CPU and memory usage of the state machine process for the duration of the mission were also recorded. All values were recorded from the *ps* command in Linux. The *SMACC* state machine process was executing in a Ryzen 7 5800X CPU, with a clock frequency locked at 3.8 GHz.

Figure 7.2: Turtle Hop mission state machine graph, recorded from *SMACC Viewer*.

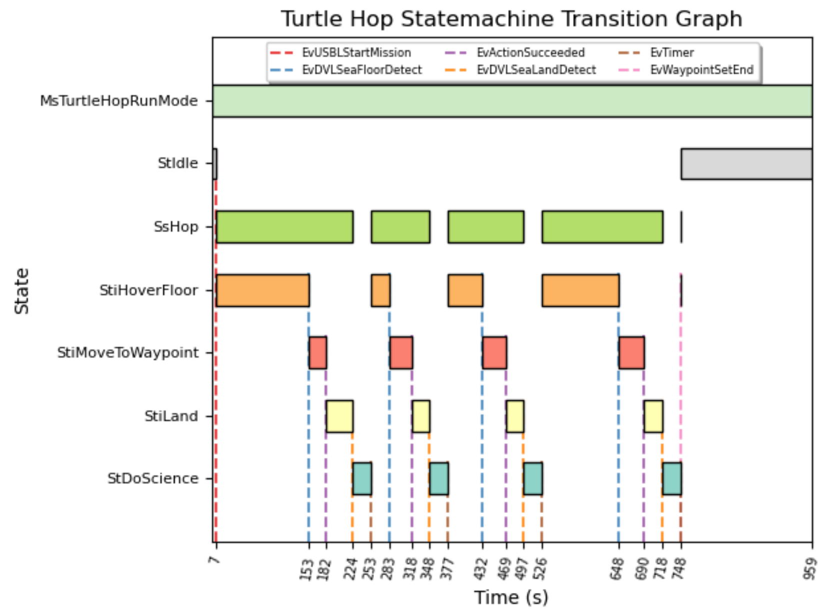


Figure 7.3: Turtle Hop mission state machine transitions graph.

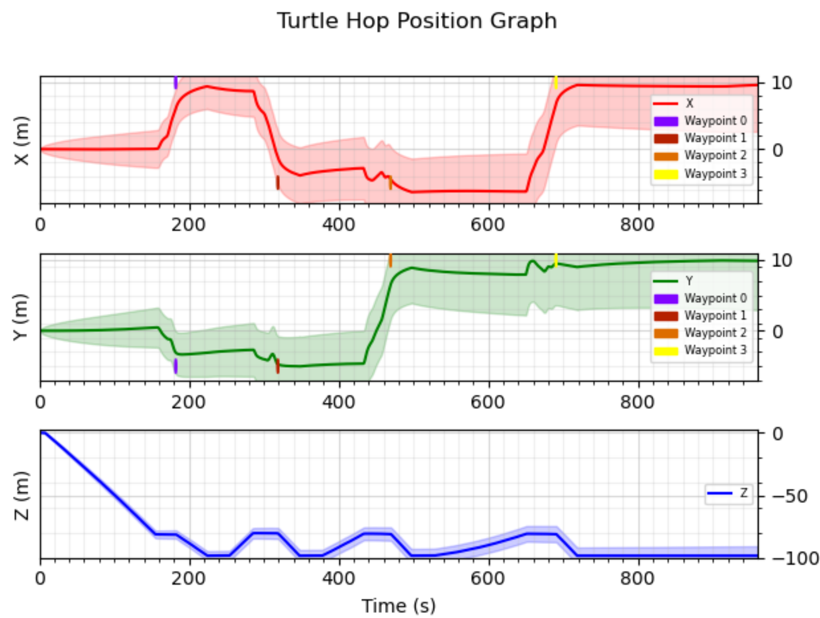


Figure 7.4: Turtle Hop mission position graph. The region around each line represents the uncertainty of the position coordinate and the vertical length of each waypoint represents their radius of acceptance.

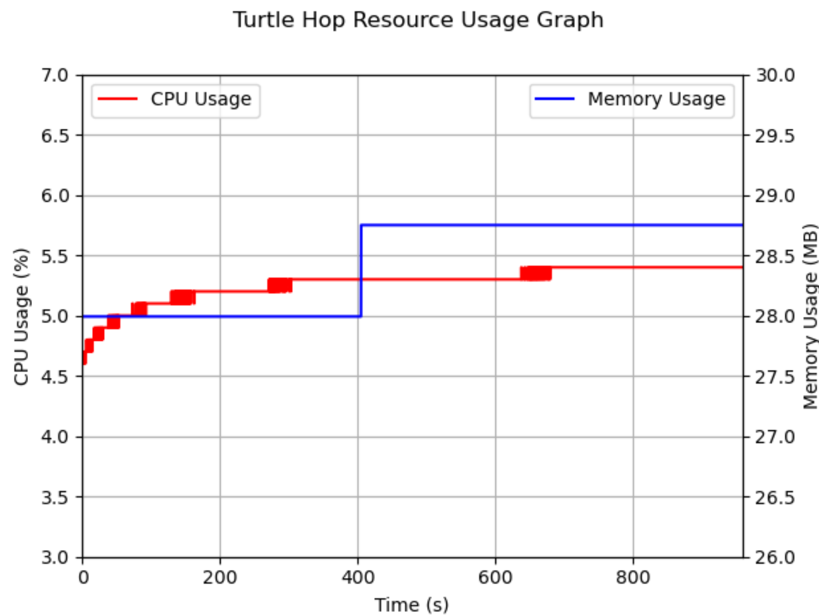


Figure 7.5: Turtle Hop mission resource usage graph.

7.2 Two Turtle Hop Coordination mission

A coordination mission between two AUVs was also designed. It's main objective was to demonstrate the capability of a configuration with cooperation between multiple Turtle 3 AUVs, in a scenario where an extended underwater communication network is desirable.

The two turtle hop mission is described as:

1. Both AUVs start at the surface of the sea and wait for a start command.
2. Each AUV moves to a waypoint on the sea floor, read from a previously configured list for each one;
3. When one AUV lands, communicate with the other AUV;
4. When a signal is received after landing, move to the next waypoint;
5. Repeat points three and four until all waypoints have been reached.

In Figure 7.6, the previously described mission is shown. The trajectory of each vehicle between waypoints is coloured equally to the numerical description for the expected sequence of states.

Two instances of the same state machine, shown in Figure 7.9, were launched, one for each of the simulated robots. In Figure 7.7, both AUVs are shown during the mission in the Gazebo simulator.

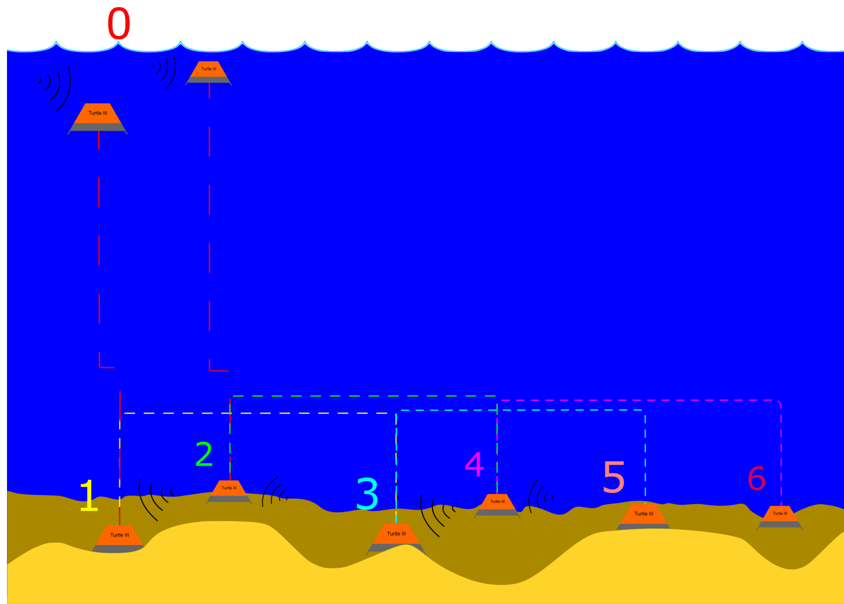


Figure 7.6: Two Turtle Hop mission diagram.

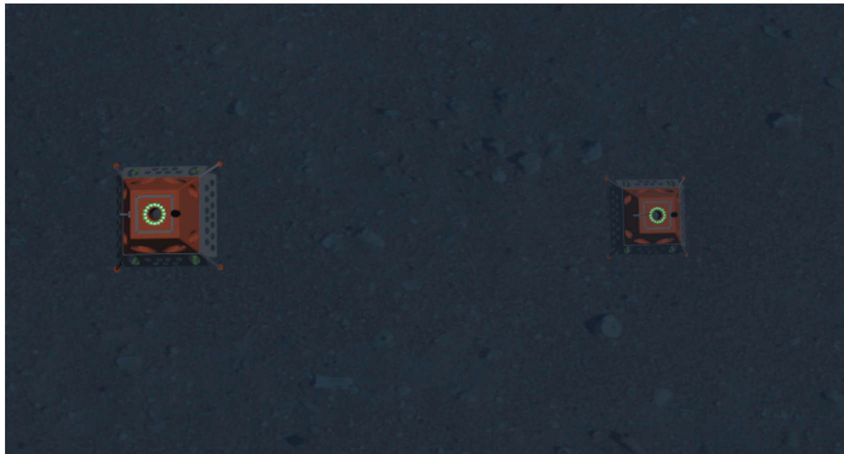


Figure 7.7: Screenshot of both Turtle 3 AUVs in the Gazebo simulator, during the Two Turtle Hop Coordination mission.

This mission reuses the *StIdle* state and *SsHop* super state in the single turtle hop mission. However, unlike the prior state machine, after each landing, the AUV enters the *StWaitForPair* state. In this state, the ready command is sent to the accompanying AUV and a transition to another state is made if a ready command is received, back to the *SsHop* super state, or if the waypoint list index is at the end, back to the *StIdle* state. These transitions can be observed in Figure 7.10a and Figure 7.10b, for each Turtle 3.

The waypoints selected for each robot are the following: (3, 6), (6, 12), (9, 18) and (12, 24). Their location is relative to the starting position of the AUVs. The position and waypoint graph for each of the robots can be seen in Figure 7.11a and Figure 7.11b.

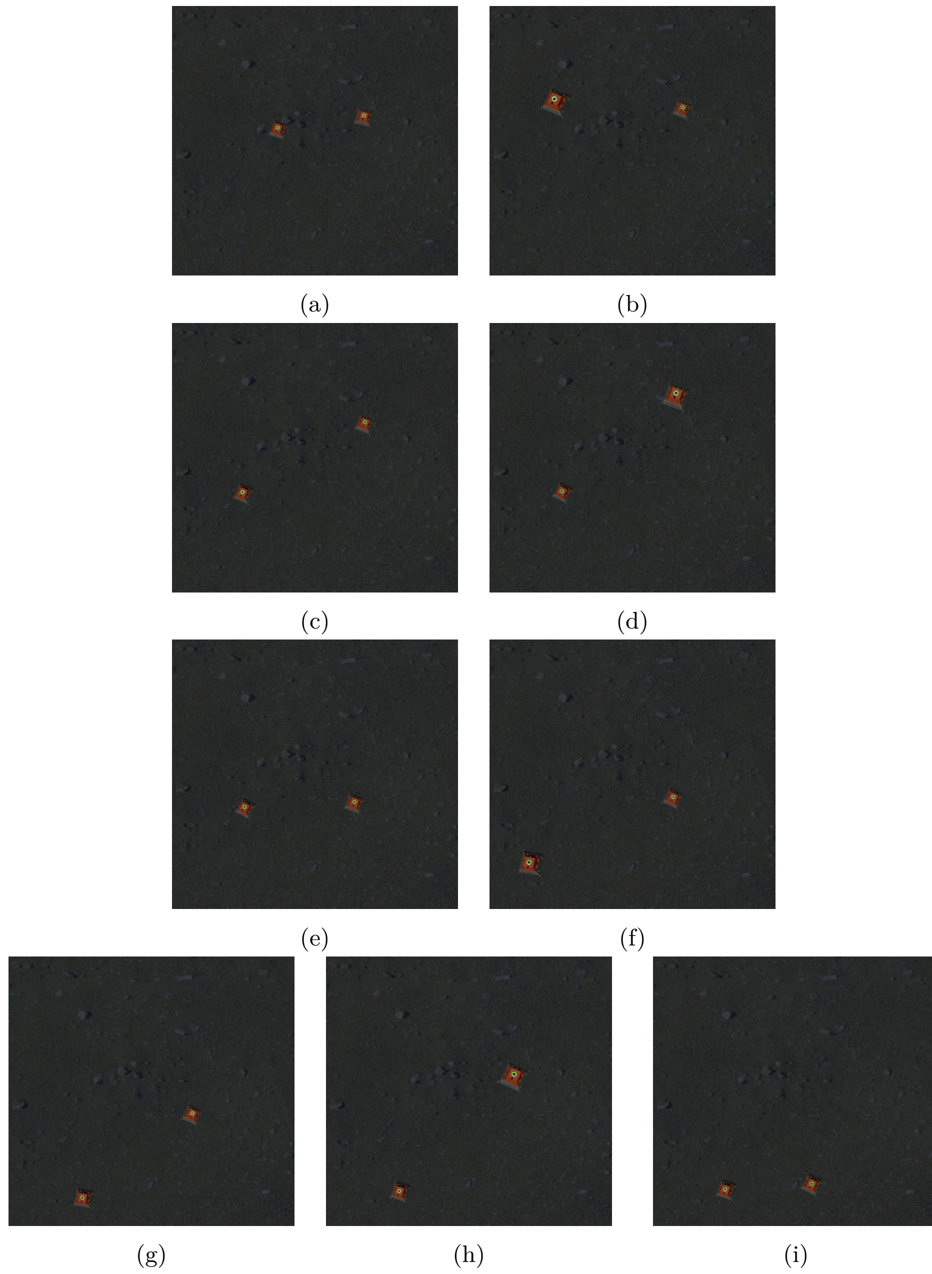
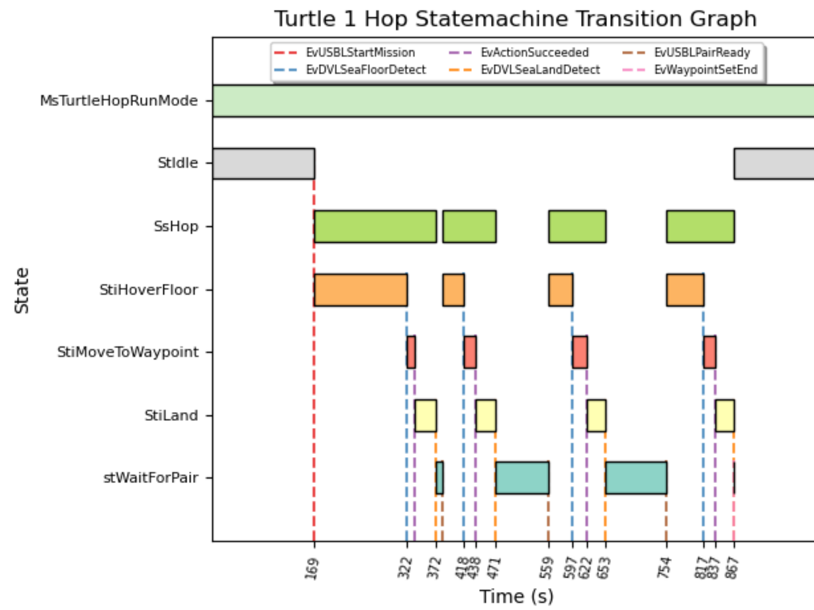
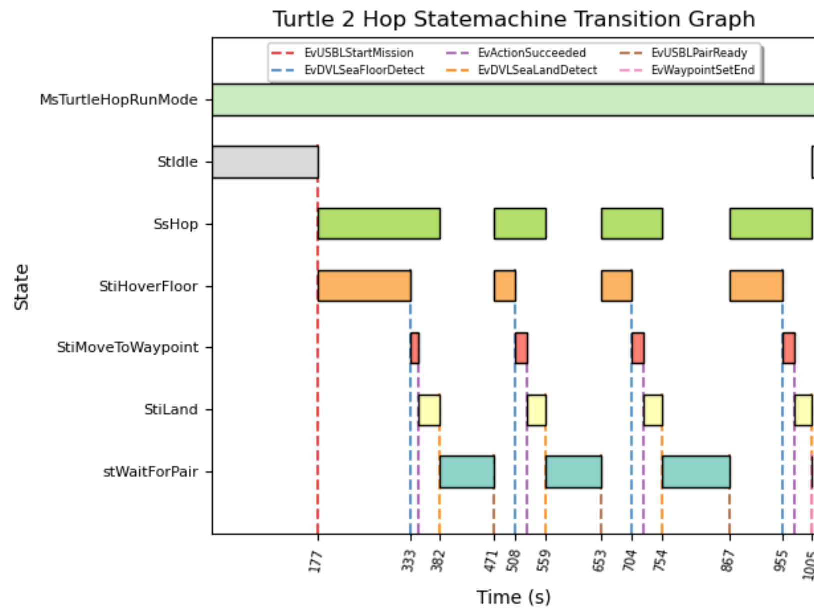


Figure 7.8: Screenshots of the Two Turtle Hop Coordination mission in the Gazebo simulator. In Figures 7.8a,7.8c,7.8e,7.8g,7.8i both Turtle 3 are landed on a waypoint. On Figures 7.8b,7.8d,7.8f,7.8h each Turtle 3 is moving towards the next waypoint interleaved with each other.

Figure 7.9: Two Turtle Hop Coordination mission state machine graph, recorded from *SMACC* Viewer.

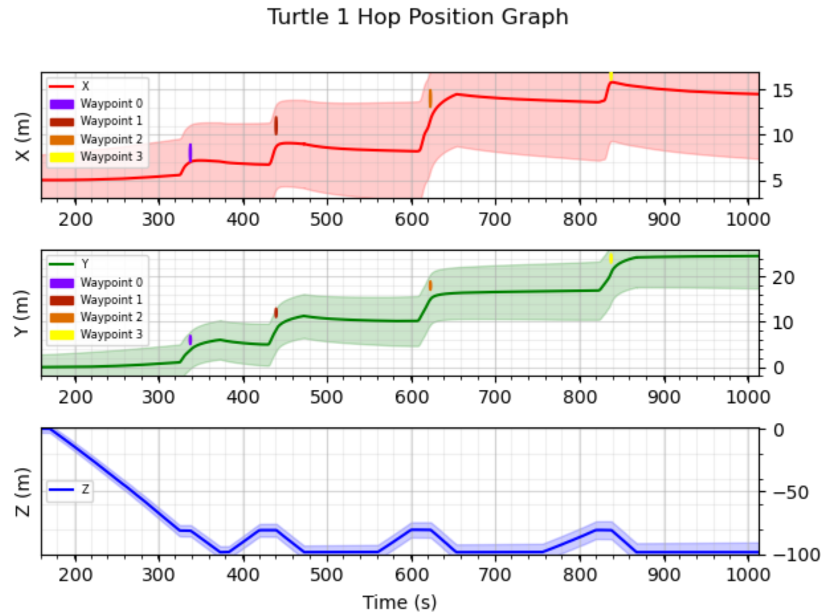


(a) Turtle 1 state machine transitions graph.

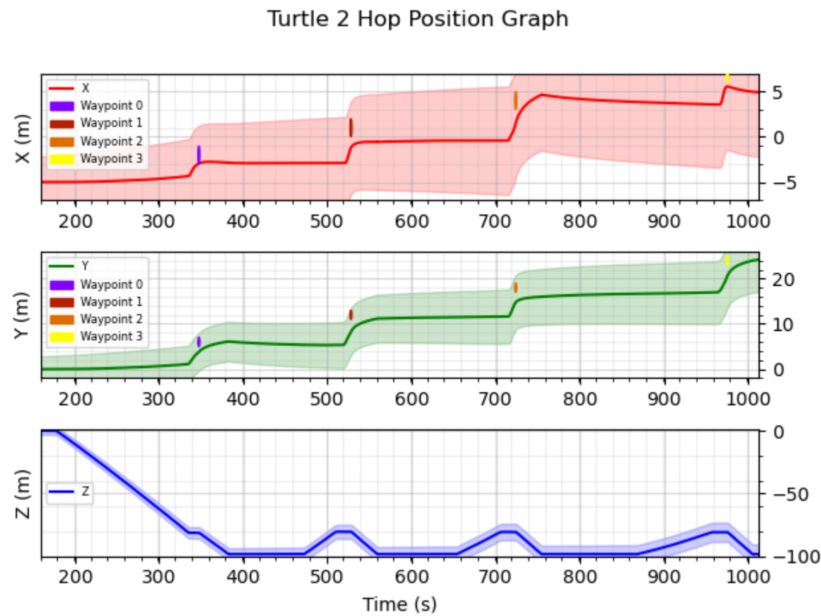


(b) Turtle 2 state machine transitions graph.

Figure 7.10: Two Turtle Hop Coordination mission state machines transitions graphs.



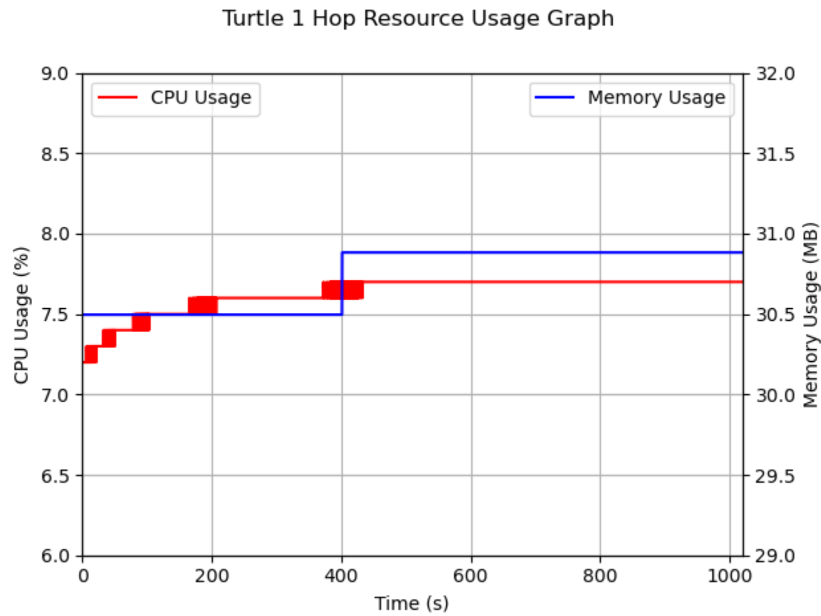
(a) Turtle 1 position graph.



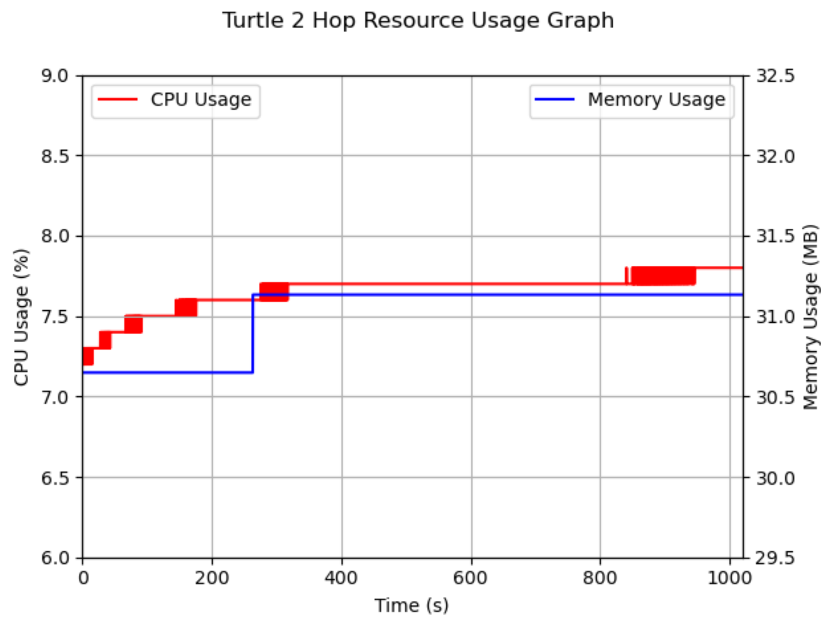
(b) Turtle 2 position graph.

Figure 7.11: Two Turtle Hop Coordination mission position graphs. The region around each line represents the uncertainty of the position coordinate and the vertical length of each waypoint represents their radius of acceptance.

The computational costs for each of the state machine processes were also analysed using the same configuration, and are shown in Figure 7.12a and Figure 7.12b.



(a) Turtle 1 resource usage graph.



(b) Turtle 2 resource usage graph.

Figure 7.12: Two Turtle Hop Coordination mission resource usage graph.

7.3 Turtle Low Battery Emergency mission

The final design and simulation is a low battery recharging mission. It involves a docking station composed of a buoy with batteries and a small tethered AUV. The small AUV has a limited range due to the power cable connecting it to the buoy. The Turtle recovery mission is detailed as:

1. The Turtle 3 detects low battery and requests to dock in the station;
2. After the station accepts the request, move within the working range of the tethered AUV;
3. When in position and stationary, alert the docking station;
4. Wait for the dock closing signal, after the small AUV reaches the docking mechanism;
5. After closing the dock, start charging;
6. When the charging of the batteries is complete, open the dock and warn the docking station;
7. Wait for the final signal of the docking station, after the smaller AUV has moved away.

The mission described above is represented in Figure 7.13. The trajectory of each vehicle between waypoints is coloured equally to the numerical description for the expected sequence of states. In states 1 and 3, the vehicle warns the docking station that it has reached the docking position and that the battery is fully charged, respectively. The buoy is submerged on average 10 m and the target docking depth is about 15 m.

The developed state machine for this mission is shown in Figure 7.14. The simulated mission starts with the Turtle 3 AUV at the sea floor, in a run mode state. In this state, a client behaviour is monitoring the battery percentage; when the battery drops under a specified value, an event is generated as soon as it is detected, and a transition is done from the run mode state to the recovery mode state. The entry state in this mode state is the *StSendWaitStart* state, which requests the docking station to dock. This behaviour is repeated until the dock responds and accepts, thereafter moving to the *StMoveToDock* state. In this state a trajectory is generated, taking into consideration the detected direction of the USBL communication to the dock, and is shown in Figure 7.15a. This trajectory, similarly to the previous missions, marks a first waypoint some meters directly above the landing position, in order to avoid collisions when moving laterally towards the final waypoint. This movement can be seen in the position graph for the Turtle 3 shown in Figure 7.17.

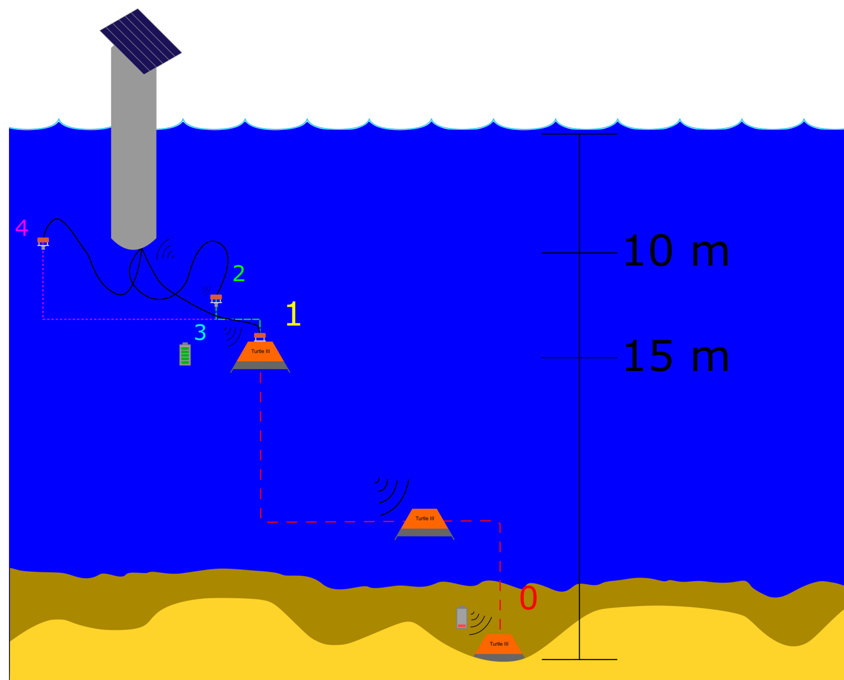


Figure 7.13: Turtle recovery mission diagram

After the docking position is reached, the state machine transitions to the *StSlowDown* state, in which the total speed is checked to make sure the AUV has slowed down enough to allow the docking. When this speed is reached, a transition is made to the *StSendReadyToDock*, which alerts the docking station to move the tethered AUV to the docking mechanism on top of the Turtle 3. When this position is reached, the buoy alerts the Turtle 3 to close the docking mechanism, moving in the *StCloseDock* state. After this action is complete, the charging of the batteries starts. This event can be seen in Figure 7.18, which shows the battery percentage over the duration of the mission. When a defined battery percentage is reached, an event is posted and a transition is made to the *StChargeDone* state. In this state, the docking station is warned that the batteries are charged; when a reply to open the dock is received, the state machine transitions to the *StOpenDock* state. After the dock is open, the Turtle 3 AUV waits for the final response of the docking station, signalling that the docking mechanism has been cleared. When this message is received, the state machine transitions out of the recovery mode and into it's previous state in the initial mode state. The state transition graph for the mission is presented in Figure 7.19.

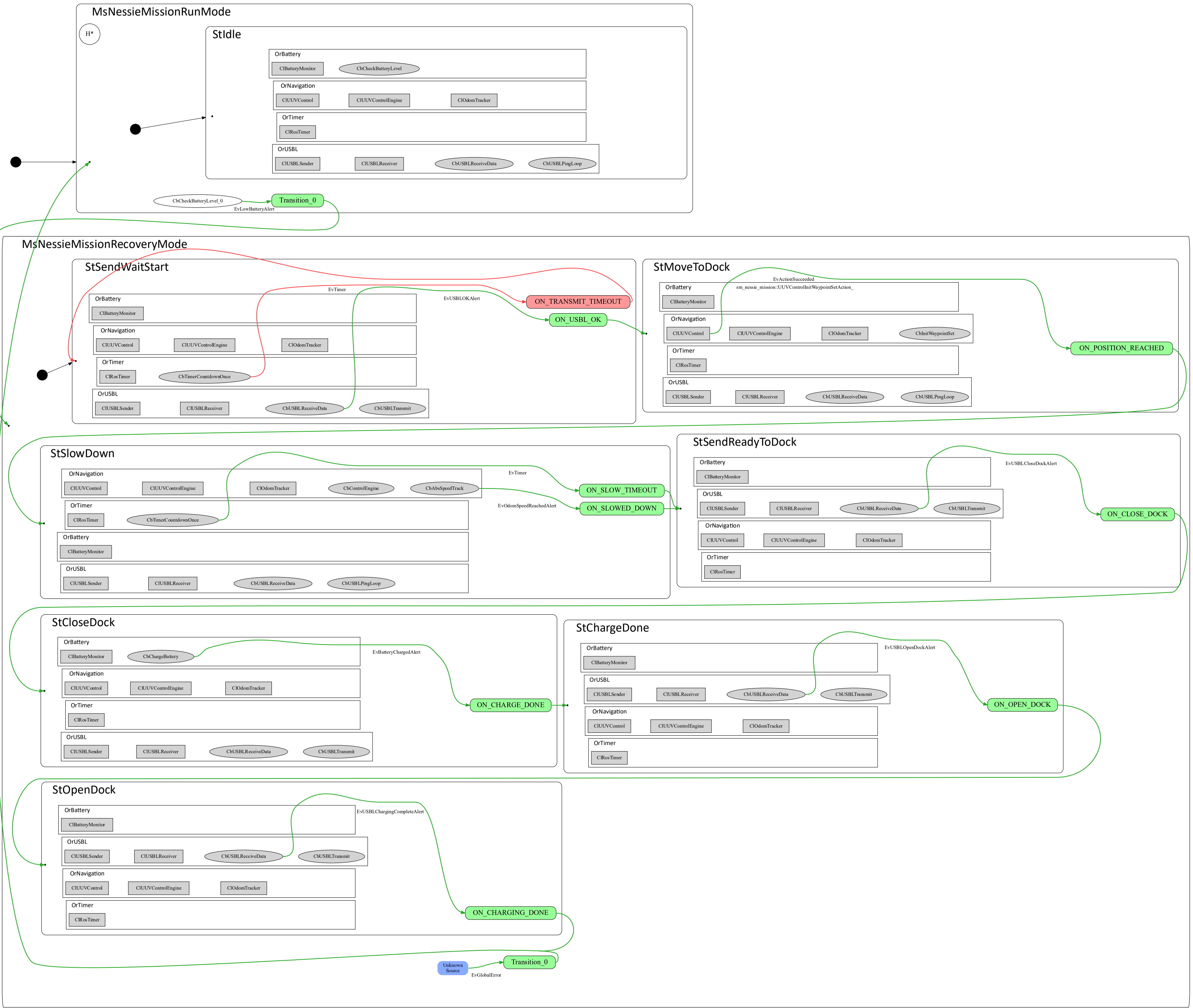
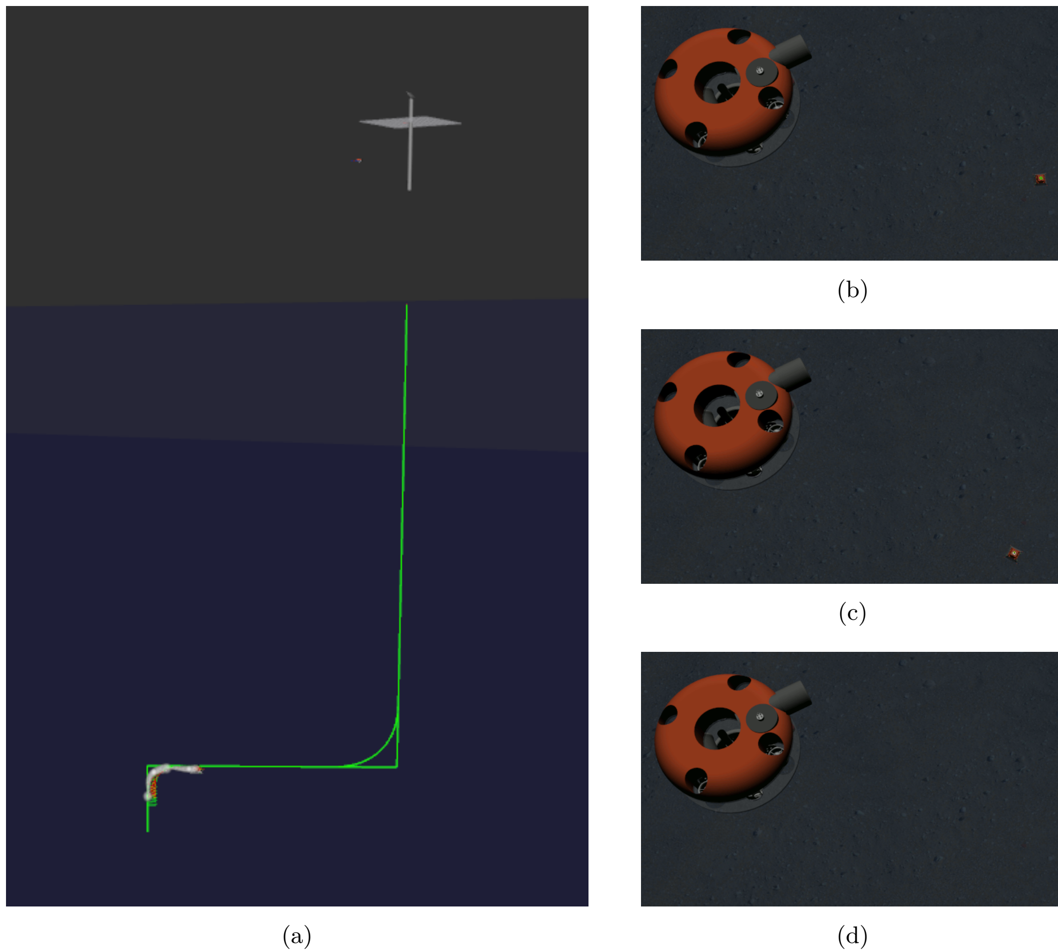


Figure 7.14: Turtle recovery mission state machine graph, recorded from *SMACC* Viewer.



(a) (b) (c) (d)

Figure 7.15: In Figure 7.15a, a screenshot in RVIZ of the generated trajectory for the Turtle 3 AUV's movement to the docking location during the Turtle recovery mission is shown. On the right side, the screenshots in the Gazebo simulator show a sequence of the Turtle 3's movement from the starting position in Figure 7.15b, until it is below the mobile dock and unseen, in Figure 7.15d.

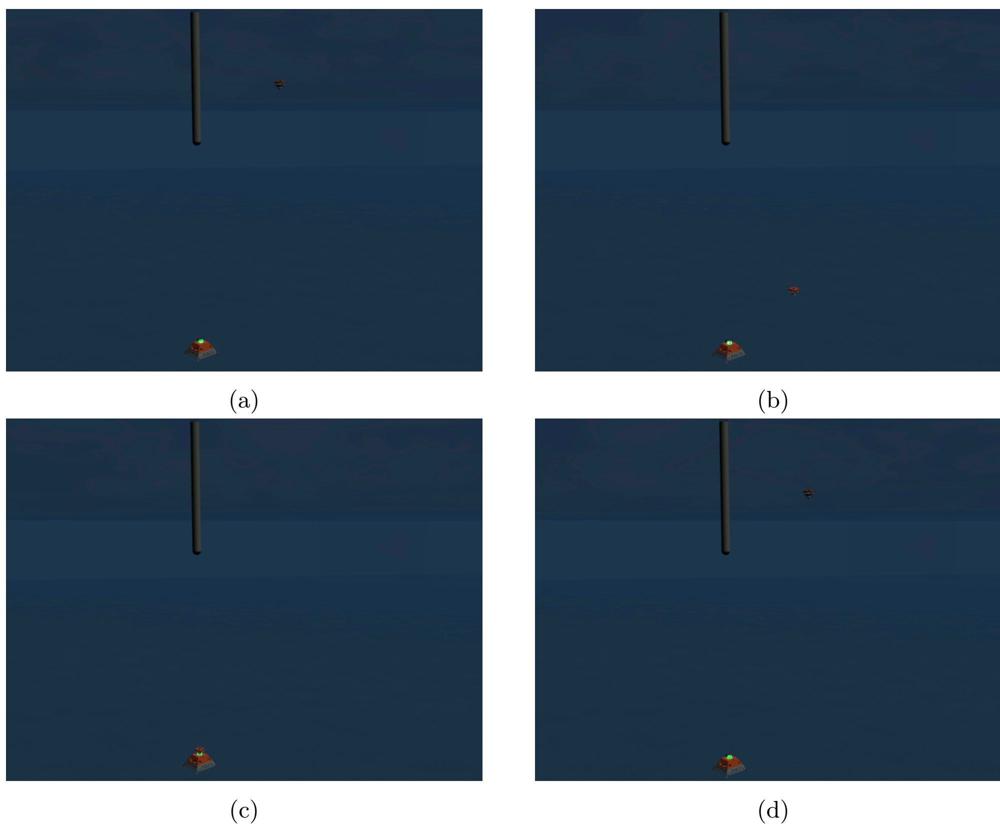


Figure 7.16: Screenshots of the Turtle recovery mission docking in the Gazebo simulator. The Figure 7.16a shows the starting position of both vehicles the instant that the docking manoeuvre starts. In Figure 7.16b, the mobile dock is approaching the Turtle 3 and in Figure 7.16c they have docked. Figure 7.16d shows the mobile dock in the safe position after undocking.

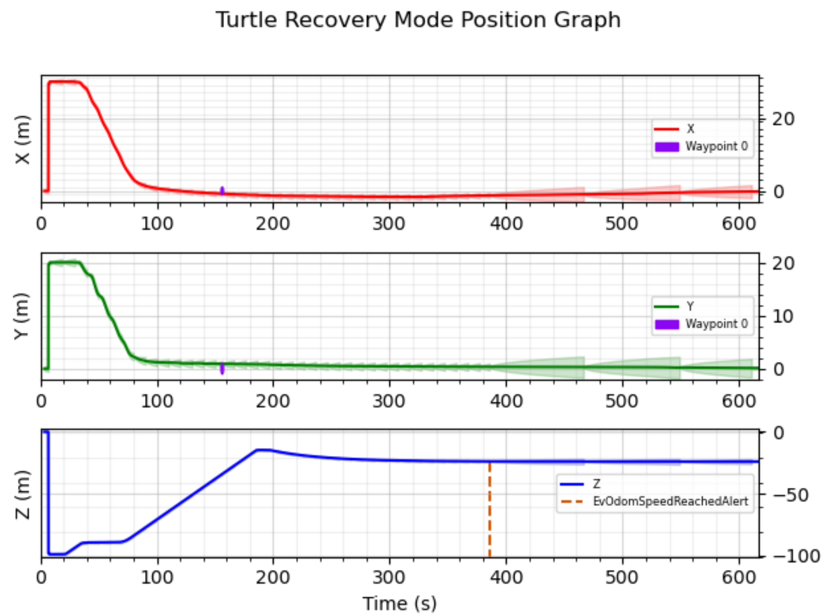


Figure 7.17: Turtle recovery mission position graph. The region around each line represents the uncertainty of the position coordinate and the vertical length of each waypoint represents their radius of acceptance.

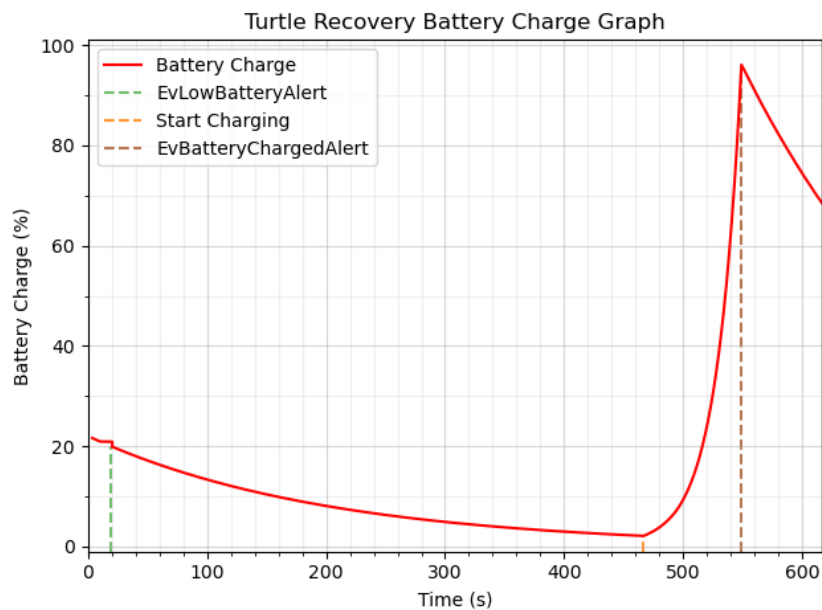


Figure 7.18: Turtle recovery mission battery charge graph.

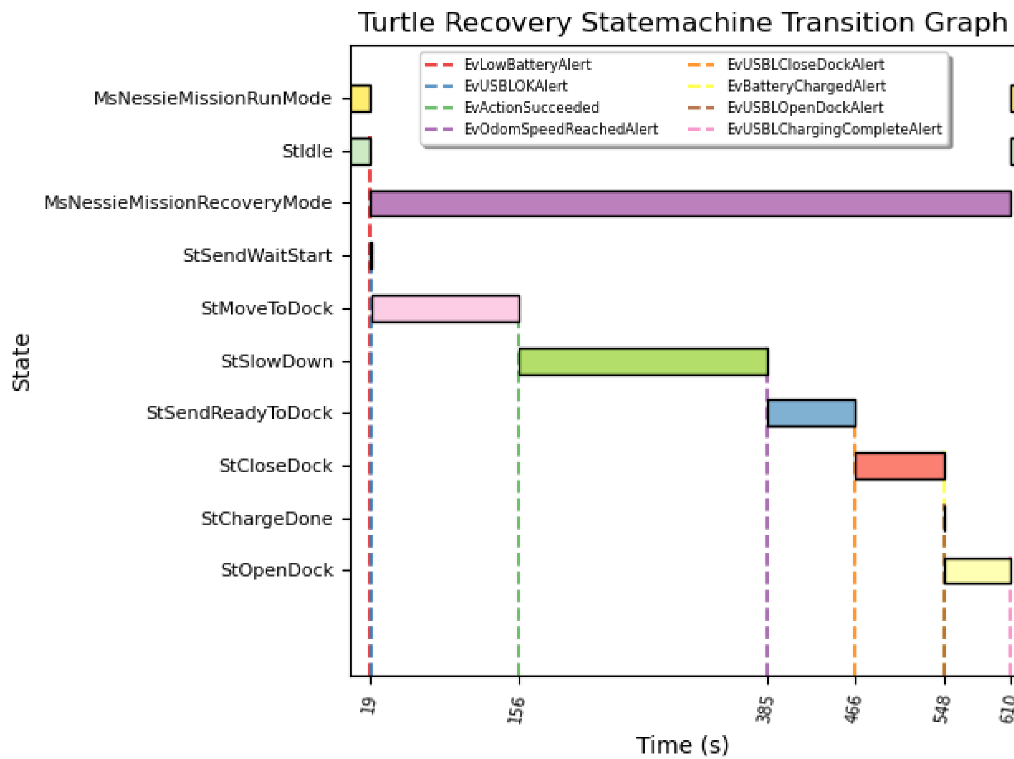


Figure 7.19: Turtle recovery mission state machine graph, recorded from *SMACC* Viewer.

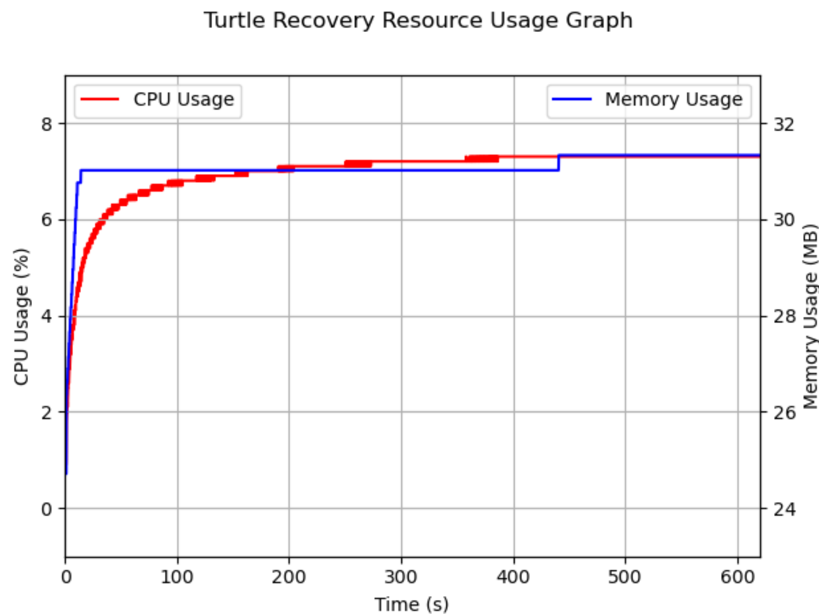


Figure 7.20: Turtle recovery mission resource usage graph.

The resource usage of this state machine process can be seen in the Figure 7.20, using the same configuration used in the previous missions.

Taking into consideration all the resource graph from the system that was used to perform the simulation, it is possible to say that the computational and memory usage within the missions will have a low impact on the on-board computer of the Turtle 3 AUV. This is due to the low amount of consistent function calls in the *SMACC* architecture, as it is event based, and also that the memory requirements don't change much between different missions due to the similar orthogonal setup, which is expected in a deployment on the physical system.

7.4 System Autonomy

Using the autonomy metrics described in Section 4.2, the implemented system has:

- A MAP score of M3, A-, P3 or M3, A4, P3 for the low battery mission;
- A Draper score of 3 for situational awareness, 2 for task planning and 3 for mobility control;
- A ACL score of 9 for Perception/Situational Awareness, 8 for Analysis/Decision Making and 9 for Communication/Cooperation;
- A Sheridan score of 10.

The attributed MAP scores represent mobility in three dimensions, being battery operated (A-) or having a planned tactic for extracting and storing energy (A4), and having tactical behaviour when a critical situation occurs.

The Draper score constitutes the usage of integrated multi-sensor fusion, the interpretation of goals to actions, and the integration of multiple actions.

The ACL score reflects the ability of awareness and interaction with other vehicles, continuous trajectory evaluation and planning, and cooperation with other vehicles and independent supervision and control.

From these metrics, it is only the Sheridan's result that does not provide much value. While individually, the MAP, Draper and ACL values have some limitations, as stated in Section 4.2, when used together they allow a more insightful observation of the system's level of autonomy.

Chapter 8

Conclusion

There was a necessity for the development of a solution that would allow the creation and execution of missions for deep seafloor environments and the control of the complex AUV required to access them. This scenario offers many challenges to a mission designer, as the AUV must be robust and reliable and must be able to complete the mission objectives. The work developed in this thesis addresses the problem of the creation and execution of a mission for complex AUV such that it is possible to make use of all of the robot's systems, it is robust and reliable, reduces the development time and complexity of the design, allows easy integration of new sub-systems and works with existing autonomous robots.

Research in the literature of control of autonomous vehicles was conducted, identifying the state of the art with regard to mission coordinator architectures, as well as robot level of autonomy metrics and ROS mission coordinators.

After the consideration of the proposed requirements, the selection of a compatible, Hierarchical Finite State Machines (HFSMs) based, mission coordinator framework in ROS, and the development of clients that interface with existing sub-systems, namely those associated with the simulator used for testing, a system that tries to fulfil all of the aforementioned objectives was created.

This system was then applied to a simulated model of the Turtle AUV in the Gazebo simulator, using Fossen's equations of motion with parameters that were obtained from CFD simulations of the vehicle's model. These simulations also considered the thrust force that the motors can apply as well as the uncertainties relating

to the sensors. These uncertainties were used for the sensor fusion in the navigation of the vehicle.

Taking into account the results that were obtained, after the testing in three different mission scenarios, it was possible to conclude that the system is capable of creating and executing missions with many interacting sub-systems, and allows for routines to be executed in response to emergencies. It was also shown that the resource usage for the execution of this system is well within the expected existing hardware in these types of vehicles. The level of autonomy was calculated under the MAP, Draper, ACL and Sheridan scales and using their combined scores it is possible to assert that the implemented solution is autonomous, capable of moving towards mission objectives while avoiding obstacles, able to react to emergencies and can cooperate with other robots.

An important topic that arose from the development of this system was the elevated amount of work required in the initial mission design, due to the lack of existing clients to interface with the necessary sub-systems. This problem was alleviated in the subsequent missions as a client interface had been created for most of the sub-systems. Another set back to the selected mission coordinator was the necessity to develop each mission as a C++ ROS package, where each of it's components is usually defined in a header file.

This thesis presents contributions in the development of mission coordinators for Lander type AUVs, allowing the use of the full potential of a vehicle, without being unmanageable to the mission designer, and allowing for complex missions while also providing tools for the assurance of the safety of the AUV.

In conclusion, it is possible to say that the objectives proposed for this thesis were reached and that the developed solution aims to meet the considered requirements.

Part of the work developed in this thesis was submitted to the OCEANS 2022 Hampton Roads conference.

8.1 Future Work

In regards to future works, the development of a mission can be greatly improved, by the creation of a conversion tool, that would allow a simpler description language to be used for the design of the mission, and subsequently be converted to the necessary folder and file structure. This would allow for swifter and more efficient mission creation.

Finally, the system can be expanded to other vehicle types, requiring only the creation of the necessary clients for each sub-system. This task can be completed, firstly, in other AUVs, as most of their systems are similar or equal to a Lander, but can be expanded to any other type of robot.

References

[1] E. Silva, A. Martins, J. M. Almeida, H. Ferreira, A. Valente, M. Camilo, A. Figueiredo, and C. Pinheiro, “Turtle - a robotic autonomous deep sea lander,” in *OCEANS 2016 MTS/IEEE Monterey*, 2016, pp. 1–5. [Cited on page 1]

[2] J. H. V. Schuppen, “What is coordination control?” *Lecture Notes in Control and Information Sciences*, vol. 456, pp. 99–106, 7 2015. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-10407-2_12 [Cited on page 7]

[3] J. Komenda and T. Masopust, “Supervisory control of discrete-event systems,” *Lecture Notes in Control and Information Sciences*, vol. 456, pp. 129–136, 2015. [Cited on pages 8 and 9]

[4] W. Wonham, K. Cai, and K. Rudie, “Supervisory control of discrete-event systems: A brief history – 1980-2015,” 07 2017. [Cited on page 9]

[5] C. G. Cassandras and S. Lafortune, “Introduction to discrete event systems,” *Introduction to Discrete Event Systems*, pp. 1–771, 2008. [Cited on page 9]

[6] J. Komenda and T. Masopust, “Coordination control of distributed discrete-event systems,” *Lecture Notes in Control and Information Sciences*, vol. 456, pp. 137–144, 2015. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-10407-2_17 [Cited on page 10]

[7] M. Yannakakis, “Hierarchical state machines,” in *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 315–330. [Cited on page 11]

[8] A. Kurt and Ümit Özgüner, “Hierarchical finite state machines for autonomous mobile systems,” *Control Engineering Practice*, vol. 21, pp. 184–194, 2 2013. [Cited on pages 11, 12, and 13]

[9] V. Sklyarov and I. Skliarova, “Hierarchical specification and design of control systems in robotics,” in *Proceedings of the 3rd Int. Conf. on Autonomous Robots and Agents-ICARA*, 2006, pp. 623–628. [Cited on pages 12 and 13]

[10] V. de Araujo, A. P. G. S. Almeida, C. T. Miranda, and F. de Barros Vidal, "A parallel hierarchical finite state machine approach to uav control for search and rescue tasks," in *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, vol. 01, 2014, pp. 410–415. [Cited on pages 12 and 13]

[11] H. Xu, Y. Zhang, and X. Feng, "Research on the decentralized supervisory control of autonomous underwater vehicle," in *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*, vol. 6, 2004, pp. 4909–4913 Vol.6. [Cited on page 12]

[12] T. Y. Teck, M. Chitre, and P. Vadakkepat, "Hierarchical agent-based command and control system for autonomous underwater vehicles," in *2010 International Conference on Autonomous and Intelligent Systems, AIS 2010*, 2010, pp. 1–6. [Cited on page 12]

[13] T. O. Fossum, P. Norgren, I. Fer, F. Nilsen, Z. C. Koenig, and M. Ludvigsen, "Adaptive sampling of surface fronts in the arctic using an autonomous underwater vehicle," *IEEE Journal of Oceanic Engineering*, vol. 46, no. 4, pp. 1155–1164, 2021. [Cited on page 12]

[14] X. Xiang, G. Xu, Q. Zhang, Z. Xiao, and X. Huang, "Coordinated control for multi-auv systems based on hybrid automata," in *2007 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2007, pp. 2121–2126. [Cited on page 12]

[15] I. Bychkov, A. Davydov, M. Kenzin, N. Maksimkin, N. Nagul, and S. Ul'yanov, "Hierarchical control system design problems for multiple autonomous underwater vehicles," in *2019 International Siberian Conference on Control and Communications (SIBCON)*, 2019, pp. 1–6. [Cited on page 12]

[16] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and AI," *CoRR*, vol. abs/2005.05842, 2020. [Online]. Available: https://arxiv.org/abs/2005.05842 [Cited on pages 12, 13, and 14]

[17] C. I. Sprague, Özkahraman, A. Munafo, R. Marlow, A. Phillips, and P. Ögren, "Improving the modularity of auv control systems using behaviour trees," in *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, 2018, pp. 1–6. [Cited on pages 12 and 14]

[18] M. Colledanchise and L. Natale, "On the implementation of behavior trees in robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5929–5936, 2021. [Cited on page 12]

[19] ——, "Handling concurrency in behavior trees," *CoRR*, vol. abs/2110.11813, 2021. [Online]. Available: https://arxiv.org/abs/2110.11813 [Cited on page 12]

[20] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Universität Hamburg, 1962. [Cited on page 12]

[21] P. Lima, H. Gracio, V. Veiga, and A. Karlsson, "Petri nets for modeling and coordination of robotic tasks," in *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.98CH36218)*, vol. 1, 1998, pp. 190–195 vol.1. [Cited on page 12]

[22] G. Yasuda, "Modeling, simulation and distributed control of robotic systems using petri net based multitask processing," in *SICE Annual Conference 2011*, 2011, pp. 1944–1949. [Cited on page 12]

[23] Z. Chang, M. Fu, Z. Tang, and H. Cai, "Autonomous mission management for an unmanned underwater vehicle," in *IEEE International Conference Mechatronics and Automation, 2005*, 02 2005, pp. 1455–1459 Vol. 3. [Cited on page 12]

[24] X. Feng, Q. Liu, and Z. Wang, "Auv modeling and analysis using a colored object-oriented petri net," in *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, vol. 2, 2006, pp. 405–409. [Cited on pages 12 and 14]

[25] N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre, "Using petri nets to specify and execute missions for autonomous underwater vehicles," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, 11 2009, pp. 4439 – 4444. [Cited on page 12]

[26] A. M. Anvar, "The navigational control of autonomous underwater vehicle (auv) using color petri nets," in *2009 4th IEEE Conference on Industrial Electronics and Applications*, 2009, pp. 2834–2840. [Cited on pages 12 and 14]

[27] X. Bian, T. Chen, Z. Yan, and Z. Qin, "Autonomous mission management and intelligent decision for auv," in *2009 International Conference on Mechatronics and Automation*, 2009, pp. 2101–2106. [Cited on page 12]

[28] N. Palomeras, P. Ridao, C. Silvestre, and A. El-fakdi, "Multiple vehicles mission coordination using petri nets," in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 3531–3536. [Cited on page 12]

[29] B. Lacerda and P. U. Lima, "Petri net based multi-robot task coordination from temporal logic specifications," *Robotics and Autonomous Systems*, vol. 122, p. 103289, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889019302441 [Cited on page 12]

[30] H. Costelha and P. Lima, "Modelling, analysis and execution of robotic tasks using petri nets," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007, pp. 1449–1454. [Cited on page 12]

[31] C. Lin and Y. Li, "Study on mission reachability problem for multiple auvs based on object-oriented petri net," in *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, 2015, pp. 1259–1264. [Cited on page 12]

[32] A. Atyabi, S. MahmoudZadeh, and S. Nefti-Meziani, "Current advancements on autonomous mission planning and management systems: An auv and uav perspective," *Annual Reviews in Control*, vol. 46, pp. 196–215, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367578818300257 [Cited on pages 15 and 16]

[33] B. Hasslacher and M. W. Tilden, "Living machines," *Robotics and Autonomous Systems*, vol. 15, no. 1, pp. 143–169, 1995, the Biology and Technology of Intelligent Autonomous Agents. [Online]. Available: https://www.sciencedirect.com/science/article/pii/092188909500019C [Cited on page 15]

[34] M. E. Cleary, M. Abramson, M. B. Adams, and S. Kolitz, "Metrics for embedded collaborative intelligent systems," *NIST SPECIAL PUBLICATION SP*, pp. 295–301, 2001. [Cited on page 16]

[35] B. T. Clough, "Metrics, schmetrics! how the heck do you determine a uav's autonomy anyway," Air Force Research Lab Wright-Patterson AFB OH, Tech. Rep., 2002. [Cited on page 16]

[36] T. B. Sheridan and W. L. Verplank, "Human and computer control of undersea teleoperators," Massachusetts Inst of Tech Cambridge Man-Machine Systems Lab, Tech. Rep., 1978. [Cited on page 16]

[37] J. Bohren, "smach - ros wiki." [Online]. Available: http://wiki.ros.org/smach [Cited on pages 17 and 20]

[38] M. Carroll and J. Perron, "actionlib - ros wiki." [Online]. Available: http://wiki.ros.org/actionlib [Cited on page 17]

[39] J. Bohren, "smach_viewer - ros wiki." [Online]. Available: http://wiki.ros.org/smach_viewer [Cited on page 17]

[40] C. Pradalier, "A task scheduler for ROS," UMI 2958 GeorgiaTech-CNRS, Research Report, Jan. 2017, rOBOTICS. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01435823 [Cited on pages 18 and 21]

[41] D. Lu, M. Ferguson, and A. Hoy, "move_base - ros wiki." [Online]. Available: http://wiki.ros.org/move_base [Cited on page 18]

[42] B. Aldrich and P. Blasco, "Smacc – state machine asynchronous c++." [Online]. Available: https://smacc.dev/ [Cited on pages 18, 19, and 21]

[43] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642387900359 [Cited on page 18]

[44] M. Steinbrink, P. Koch, S. May, B. Jung, and M. Schmidpeter, "State Machine for Arbitrary Robots for Exploration and Inspection Tasks," in *Proceedings of the 2020 4th International Conference on Vision, Image and Signal Processing.* New York, NY, USA: ACM, dec 2020, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3448823.3448857 [Cited on pages 19 and 21]

[45] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, April 2017. [Cited on pages 19 and 21]

[46] V. A. Ziparo, L. Iocchi, and D. Nardi, "Petri net plans," in *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, 2006, pp. 267–290. [Cited on pages 19, 20, and 21]

[47] P. F. Palamara, V. A. Ziparo, L. Iocchi, D. Nardi, and P. Lima, "Teamwork design based on petri net plans," in *Robot Soccer World Cup.* Springer, 2008, pp. 200–211. [Cited on pages 19 and 21]

[48] D. Sytnyk, "Ambiente de simulação para o sistema de exploração robótica subaquática unexmin," Master's thesis, 2018. [Online]. Available: http://hdl.handle.net/10400.22/12431 [Cited on page 23]

[49] M. M. M. Manhães, S. A. Scherer, M. Voss, L. R. Douat, and T. Rauschenbach, "UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation," in *OCEANS 2016 MTS/IEEE Monterey.* IEEE, sep 2016. [Online]. Available: https://doi.org/10.1109%2Foceans.2016.7761080 [Cited on page 23]

[50] T. I. Fossen, *Handbook of marine craft hydrodynamics and motion control.* John Wiley & Sons, 2011. [Cited on page 23]

Appendix A

Example of a mission: Turtle Hop

A.1 sm_turtle_hop

A.1.1 sm_turtle_hop.h

```
#pragma once
#include <smacc/smacc.h>

// CLIENTS
#include <ros_timer_client/cl_ros_timer.h>

// CLIENT BEHAVIORS
#include <multirole_sensor_client/client_behaviors/
    cb_default_multirole_sensor_behavior.h>
#include <sm_turtle_hop/clients/cl_usbl_receiver/client_behaviors/
    cb_usbl_receive_data.h>
...

using namespace sm_turtle_hop::cl_usbl_receiver;
using namespace sm_turtle_hop::cl_usbl_sender;
...

//STATE REACTORS
```

```
17
18 using namespace smacc::state_reactors;
19
20 // ORTHOGONALS
21 #include <sm_turtle_hop/orthogonals/or_navigation.h>
22 #include <sm_turtle_hop/orthogonals/or_usbl.h>
23 ...
24
25 namespace sm_turtle_hop
26 {
27 //SUPERSTATE FORWARD DECLARATIONS
28 namespace SS1
29 {
30     class SsHop;
31 }
32 //STATE FORWARD DECLARATIONS
33 class StIdle;
34
35 // MODE STATES FORWARD DECLARATIONS
36 class MsTurtleHopRunMode;
37 class MsTurtleHopRecoveryMode;
38
39 // custom event example
40 struct EvGlobalError : sc::event<EvGlobalError>
41 {
42 };
43 } // namespace sm_turtle_hop
44
45 using namespace sm_turtle_hop;
46 using namespace cl_ros_timer;
47 using namespace smacc;
48
49 namespace sm_turtle_hop
50 {
51 struct SmTurtleHop
52     : public smacc::SmaccStateMachineBase<SmTurtleHop,
       MsTurtleHopRunMode>
53 {
54     typedef mpl::bool_<false> shallow_history;
55     typedef mpl::bool_<false> deep_history;
56     typedef mpl::bool_<false> inherited_deep_history;
57     using SmaccStateMachineBase::SmaccStateMachineBase;
58
59     virtual void onInitialize() override
60     {
61         this->createOrthogonal<OrNavigation>();
62         this->createOrthogonal<OrUSBL>();
63         this->createOrthogonal<OrTimer>();
64         this->createOrthogonal<OrObstaclePerception>();
```

```
65      }
66  };
67  } // namespace sm_turtle_hop
68
69  //MODE STATES
70  #include <sm_turtle_hop/modestates/ms_turtle_hop_run_mode.h>
71  #include <sm_turtle_hop/modestates/ms_turtle_hop_recovery_mode.h>
72
73  //STATES
74  #include <sm_turtle_hop/states/st_idle.h>
75  #include <sm_turtle_hop/states/st_do_science.h>
76
77  //SUPERSTATES
78  #include <sm_turtle_hop/superstates/ss_hop.h>
```

A.1.2 sm_turtle_hop.cpp

```
1  #include <sm_turtle_hop/sm_turtle_hop.h>
2  int main(int argc, char **argv)
3  {
4      ros::init(argc, argv, "turtle_hop");
5      ros::NodeHandle nh;
6
7      ros::Duration(5).sleep();
8      smacc::run<SmTurtleHop>();
9  }
```

A.2 Modestates

A.2.1 ms_turtle_hop_run_mode.h

```
1  #include <smacc/smacc.h>
2
3  namespace sm_turtle_hop
4  {
5      // STATE DECLARATION
6      class MsTurtleHopRunMode : public smacc::SmaccState<
           MsTurtleHopRunMode, SmTurtleHop, StIdle>
7      {
8      public:
9          using SmaccState::SmaccState;
10
11         // TRANSITION TABLE
```

```
12          typedef mpl::list<
13                  Transition<EvGlobalError, MsTurtleHopRecoveryMode>
14          >reactions;
15      };
16  }
```

A.2.2 ms_turtle_hop_recovery_mode.h

```
1   #include <smacc/smacc.h>
2   namespace sm_turtle_hop
3   {
4       // STATE DECLARATION
5       class MsTurtleHopRecoveryMode : public smacc::SmaccState<
            MsTurtleHopRecoveryMode, SmTurtleHop>
6       {
7       public:
8           using SmaccState::SmaccState;
9
10          // TRANSITION TABLE
11          typedef mpl::list<
12                  Transition<EvGlobalError, sc::deep_history<
                        MsTurtleHopRunMode::LastDeepState>>
13          >reactions;
14      };
15  }
```

A.3 ss_hop

A.3.1 ss_hop.h

```
1   #include <smacc/smacc.h>
2   #include <tf/transform_datatypes.h>
3   #include <angles/angles.h>
4
5   namespace sm_turtle_hop
6   {
7       namespace SS1
8       {
9           namespace sm_turtle_hop
10          {
11              namespace hop_states
12              {
13                  // FORWARD DECLARATIONS OF INNER STATES
```

```
14                  class StiHoverFloor;
15                  class StiMoveToWaypoint;
16                  class StiLand;
17              }// namespace hop_states
18          }// namespace sm_turtle_hop
19
20          using namespace sm_turtle_hop::hop_states;
21
22          // STATE DECLARATION
23          struct SsHop : smacc::SmaccState<SsHop, MsTurtleHopRunMode
                , StiHoverFloor>
24          {
25          public:
26              using SmaccState::SmaccState;
27
28              ...
29
30          };
31
32          // FORWARD DECLARATION FOR THE SUPERSTATE
33          using SS = SS1::SsHop;
34          #include <sm_turtle_hop/states/hop_states/sti_hover_floor.
                h>
35          #include <sm_turtle_hop/states/hop_states/
                sti_move_to_waypoint.h>
36          #include <sm_turtle_hop/states/hop_states/sti_land.h>
37      } // namespace SS1
38  } // namespace sm_turtle_hop
```

A.3.2 sti_hover_floor.h

```
1   #define HOVER_HEIGHT 20
2
3   namespace sm_turtle_hop
4   {
5       namespace hop_states
6       {
7           // STATE DECLARATION
8           struct StiHoverFloor : smacc::SmaccState<StiHoverFloor, SS
                >
9           {
10          public:
11              using SmaccState::SmaccState;
12
13              // DECLARE CUSTOM OBJECT TAGS
14              struct MISSION_DONE : SUCCESS{};
15              struct FLOOR : SUCCESS{};
```

```
16          struct NO_FLOOR : ABORT{};
17
18          // TRANSITION TABLE
19          typedef mpl::list<
20              Transition<EvDVLSeaFloorDetect<CbDVLDetectFloor,
                    OrObstaclePerception>, StiMoveToWaypoint, FLOOR
                    >,
21              Transition<EvWaypointSetEnd<CbCheckWaypointSetEnd,
                     OrNavigation>, StIdle, MISSION_DONE>
22          >reactions;
23
24          // STATE FUNCTIONS
25          static void staticConfigure()
26          {
27              configure_orthogonal<OrNavigation, CbVelHeave>();
28              configure_orthogonal<OrNavigation,
                    CbCheckWaypointSetEnd>();
29              configure_orthogonal<OrObstaclePerception,
                    CbDVLDetectFloor>(HOVER_HEIGHT);
30          }
31
32          void runtimeConfigure()
33          {
34              ROS_INFO("runtimeConfigure");
35              // Check distance to floor
36              // If distance is -1 then no floor detected and
                    needs to dive or distance above 20m
37              // If distance is smaller than 20m than needs to
                    ascend
38              cl_dvl::ClDVL *dvlClient;
39              this->requiresClient(dvlClient);
40              auto dvlData = dvlClient->getComponent<
                    CpDVLSeaFloor>();
41              double distanceToFloor = dvlData->
                    getSeaFloorDistance();
42              auto navigationHeave = getOrthogonal<OrNavigation
                    >()
43                      ->template getClientBehavior<CbVelHeave>()
                            ;
44              ROS_INFO("distanceToFloor %lf", distanceToFloor);
45              if(distanceToFloor == 0 || distanceToFloor >
                    HOVER_HEIGHT)
46              {
47                  navigationHeave->setSpeed(-3);
48              }
49              else if (distanceToFloor <= HOVER_HEIGHT)
50              {
51                  navigationHeave->setSpeed(3);
52              }
```

```
53            }
54
55            ...
56
57        };
58    }
59 }
```

A.3.3 sti_move_to_waypoint.h

```
1  #include <geometry_msgs/Pose.h>
2
3  namespace sm_turtle_hop
4  {
5      namespace hop_states
6      {
7
8          // STATE DECLARATION
9          struct StiMoveToWaypoint : smacc::SmaccState<
               StiMoveToWaypoint, SS>
10         {
11         public:
12             using SmaccState::SmaccState;
13
14             // DECLARE CUSTOM OBJECT TAGS
15             struct ON_POSITION_REACHED : SUCCESS{};
16
17             // TRANSITION TABLE
18             typedef mpl::list<
19                 Transition<EvActionSucceeded<ClUUVControl,
                       OrNavigation>, StiLand, ON_POSITION_REACHED>
20             >reactions;
21
22             // STATE FUNCTIONS
23             static void staticConfigure()
24             {
25                 configure_orthogonal<OrNavigation,
                       CbInitWaypointSet>();
26             }
27
28             void runtimeConfigure()
29             {
30                 cl_uuv_control::ClUUVControl *uuvcontrolClient;
31                 this->requiresClient(uuvcontrolClient);
32                 auto poseData = uuvcontrolClient->getComponent<
                       Pose>();
33                 auto lastpose = poseData->toPoseStampedMsg();
```

```
34                  auto waypointNavigator = uuvcontrolClient->
                        getComponent<WaypointNavigator>();
35                  geometry_msgs::Pose nextWaypoint;
36                  nextWaypoint = waypointNavigator->getNextWaypoints
                        ();
37                  auto initWaypointSet = getOrthogonal<OrNavigation
                        >()
38                          ->template getClientBehavior<
                                CbInitWaypointSet>();
39                  std::vector<uuv_control_msgs::Waypoint> waypoints;
40                  uuv_control_msgs::Waypoint waypointOne;
41                  //Prepare waypoints
42                  //Header
43                  waypointOne.header.frame_id = "odom_turtle3";
44                  waypointOne.header.stamp = ros::Time::now();
45                  // Params
46                  waypointOne.max_forward_speed = 0.4;
47                  waypointOne.heading_offset = 0;
48                  waypointOne.radius_of_acceptance = 1;
49                  waypointOne.use_fixed_heading = true;
50                  // Point
51                  waypointOne.point.x = nextWaypoint.position.x;
52                  waypointOne.point.y = nextWaypoint.position.y;
53                  waypointOne.point.z = lastpose.pose.position.z;
54                  waypoints.push_back(waypointOne);
55                  initWaypointSet->waypointset_ = waypoints;
56              }
57
58              ...
59
60          };
61      }
62  }
```

A.3.4 sti_land.h

```
1   namespace sm_turtle_hop
2   {
3       namespace hop_states
4       {
5           // STATE DECLARATION
6           struct StiLand : smacc::SmaccState<StiLand, SS>
7           {
8           public:
9               using SmaccState::SmaccState;
10
11              // DECLARE CUSTOM OBJECT TAGS
```

```
12              struct LANDED : SUCCESS{};
13
14              // TRANSITION TABLE
15              typedef mpl::list<
16              Transition<EvDVLSeaLandDetect<CbDVLDetectLand,
                   OrObstaclePerception>, StDoScience, LANDED>
17              >reactions;
18
19              // STATE FUNCTIONS
20              static void staticConfigure()
21              {
22                  configure_orthogonal<OrNavigation, CbVelHeave>();
23                  configure_orthogonal<OrObstaclePerception,
                       CbDVLDetectLand>();
24              }
25
26              void runtimeConfigure()
27              {
28                  ROS_INFO("runtimeConfigure");
29                  // Check distance to floor
30                  // If distance is -1 then no floor detected and
                       needs to dive or distance above 20m
31                  // If distance is smaller than 20m than needs to
                       ascend
32                  auto navigationHeave = getOrthogonal<OrNavigation
                       >()
33                          ->template getClientBehavior<CbVelHeave>()
                               ;
34                  navigationHeave->setSpeed(-3);
35              }
36
37              ...
38
39          };
40      }
41  }
```

A.4 States

A.4.1 st_do_science.h

```
1  #include <smacc/smacc.h>
2
3  namespace sm_turtle_hop
4  {
5      using namespace smacc::default_transition_tags;
```

```
6     // STATE DECLARATION
7     struct StDoScience : smacc::SmaccState<StDoScience,
          MsTurtleHopRunMode>
8     {
9         using SmaccState::SmaccState;
10
11        // DECLARE CUSTOM OBJECT TAGS
12        struct ON_SCIENCE_DONE : SUCCESS{};
13
14        // TRANSITION TABLE
15        typedef mpl::list<
16                Transition<EvTimer<CbTimerCountdownOnce, OrTimer>,
                      sm_turtle_hop::SS1::SsHop, ON_SCIENCE_DONE>
17        >reactions;
18
19        // STATE FUNCTIONS
20        static void staticConfigure()
21        {
22            configure_orthogonal<OrTimer, CbTimerCountdownOnce
                  >(30); // EvTimer triggers once at 10 client ticks
                  (10 s)
23        }
24
25        ...
26
27    };
28 } // namespace sm_turtle_hop
```

A.4.2 st_idle.h

```
1  #include <smacc/smacc.h>
2
3  namespace sm_turtle_hop
4  {
5      using namespace smacc::default_transition_tags;
6      // STATE DECLARATION
7      struct StIdle : smacc::SmaccState<StIdle, MsTurtleHopRunMode>
8      {
9          using SmaccState::SmaccState;
10
11         // DECLARE CUSTOM OBJECT TAGS
12         struct ON_USBL_START : SUCCESS{};
13
14         // TRANSITION TABLE
15         typedef mpl::list<
16         Transition<EvUSBLStartMission<CbUSBLReceiveData, OrUSBL>,
               sm_turtle_hop::SS1::SsHop, ON_USBL_START>
```

```
17          >reactions;
18
19          // STATE FUNCTIONS
20          static void staticConfigure()
21          {
22              configure_orthogonal<OrUSBL, CbUSBLReceiveData>();
23          }
24
25          ...
26
27      };
28  } // namespace sm_turtle_hop
```

A.5 Orthogonals

A.5.1 or_navigation.h

```
1  #pragma once
2
3  #include <smacc/smacc_orthogonal.h>
4  #include <sm_turtle_hop/clients/cl_uuv_control/cl_uuv_control.h>
5  #include <sm_turtle_hop/clients/cl_control/cl_control.h>
6  ...
7
8  namespace sm_turtle_hop
9  {
10 using namespace cl_uuv_control;
11 using namespace cl_uuv_control_engine;
12 using namespace cl_control;
13 using namespace cl_odom_tracker;
14
15 class OrNavigation : public smacc::Orthogonal<OrNavigation>
16 {
17 public:
18     virtual void onInitialize() override
19     {
20         std::string odom_tracker_topic_name, control_topic_name;
21         std::string ns = ros::this_node::getNamespace();
22         auto uuvcontrolClient = this->createClient<ClUUVControl>("
               UUVControl_action_server_node");
23         uuvcontrolClient->initialize();
24         auto uuvcontrolEngineClient = this->createClient<
               ClUUVControlEngine>("
               UUVControl_engine_action_server_node");
25         uuvcontrolEngineClient->initialize();
26
```

```
27         // Control Client
28         control_topic_name = ns + "/cmd_vel";
29         auto controlClient = this->createClient<ClControl>(
               control_topic_name);
30         controlClient->initialize();
31 
32         // Receiver Client
33         odom_tracker_topic_name = ns + "/odometry/filtered_twist";
34         auto odomTrackerClient = this->createClient<ClOdomTracker
               >(odom_tracker_topic_name, ros::Duration(0));
35         odomTrackerClient->initialize();
36         uuvcontrolClient->createComponent<cl_uuv_control::Pose>();
37 
38         // create waypoints navigator component
39         auto waypointsNavigator = uuvcontrolClient->
               createComponent<cl_uuv_control::WaypointNavigator>();
40 
41         std::string planfilepath;
42         ros::NodeHandle nh("~");
43         if (nh.getParam("/sm_turtle_hop/waypoints_plan",
               planfilepath))
44         {
45             waypointsNavigator->loadWayPointsFromFile(planfilepath
                   );
46         }
47     }
48 };
49 
50 } // namespace sm_turtle_hop
```

A.5.2 or_obstacle_perception.h

```
1  #pragma once
2  
3  #include <smacc/smacc_orthogonal.h>
4  #include <sm_turtle_hop/clients/cl_dvl/cl_dvl.h>
5  #include <sm_turtle_hop/clients/cl_dvl/components/cp_dvl_sea_floor
       .h>
6  
7  namespace sm_turtle_hop
8  {
9      using namespace cl_dvl;
10 
11     class OrObstaclePerception : public smacc::Orthogonal<
           OrObstaclePerception>
12     {
13     public:
```

```
14          virtual void onInitialize() override
15          {
16              std::string dvl_track_topic_name;
17              ros::NodeHandle nh("~");
18              std::string ns = ros::this_node::getNamespace();
19              dvl_track_topic_name = ns + "/dvl";
20              auto dvlClient = this->createClient<ClDVL>(
                    dvl_track_topic_name, ros::Duration(0));
21              dvlClient->initialize();
22              dvlClient->createComponent<CpDVLSeaFloor>(
                    dvl_track_topic_name);
23          }
24      };
25  } // namespace sm_turtle_hop
```

A.5.3 or_timer.h

```
1  #include <smacc/smacc.h>
2  #include <ros_timer_client/cl_ros_timer.h>
3
4  namespace sm_turtle_hop
5  {
6  using namespace cl_ros_timer;
7
8  class OrTimer : public smacc::Orthogonal<OrTimer>
9  {
10 public:
11     virtual void onInitialize() override
12     {
13         auto client = this->createClient<ClRosTimer>(ros::Duration
               (1));
14         client->initialize();
15     }
16 };
17 } // namespace sm_turtle_hop
```

A.5.4 or_usbl.h

```
1  #pragma once
2
3  #include <smacc/smacc_orthogonal.h>
4  #include <sm_turtle_hop/clients/cl_usbl_sender/cl_usbl_sender.h>
5  #include <sm_turtle_hop/clients/cl_usbl_receiver/cl_usbl_receiver.
       h>
```

```
6
7 namespace sm_turtle_hop
8 {
9     using namespace cl_usbl_sender;
10    class OrUSBL : public smacc::Orthogonal<OrUSBL>
11    {
12    public:
13        virtual void onInitialize() override
14        {
15            // Sender Client
16            auto actionclient = this->createClient<ClUSBLSender>("
                  usbl_send_action_server_node");
17            actionclient->initialize();
18
19            // Receiver Client
20            std::string ns = ros::this_node::getNamespace();
21            auto usbl_data_topic_name = "/" + ns + "/usbl/data";
22
23            auto clUSBLReceiver = this->createClient<
                  cl_usbl_receiver::ClUSBLReceiver>(
                  usbl_data_topic_name, ros::Duration(0));
24            clUSBLReceiver->initialize();
25        }
26    };
27 } // namespace sm_turtle_hop
```

A.6 cl_usbl_receiver

A.6.1 cl_usbl_receiver.h

```
1 #pragma once
2
3 #include <multirole_sensor_client/cl_multirole_sensor.h>
4 #include "ros_usbl_sensor/USBLData.h"
5
6 namespace sm_turtle_hop
7 {
8     namespace cl_usbl_receiver
9     {
10        class ClUSBLReceiver : public cl_multirole_sensor::
              ClMultiroleSensor<ros_usbl_sensor::USBLData>
11        {
12        public:
13            ClUSBLReceiver(std::string topicname, ros::Duration
                  timeout)
14            {
```

```
15                  this->topicName = topicname;
16                  this->timeout_ = timeout;
17              }
18          };
19      } // namespace cl_usbl_receiver
20  } // namespace sm_turtle_hop
```

A.6.2 cb_usbl_receive_data.h

```
1   #pragma once
2
3   #include "ros_usbl_sensor/USBLData.h"
4   #include <sm_turtle_hop/clients/cl_usbl_receiver/cl_usbl_receiver.
        h>
5   #include <multirole_sensor_client/client_behaviors/
        cb_default_multirole_sensor_behavior.h>
6   #include <std_msgs/String.h>
7
8   namespace sm_turtle_hop
9   {    namespace cl_usbl_receiver
10      {
11          template <typename TSource, typename TOrthogonal>
12          struct EvUSBLStartMission : sc::event<EvUSBLStartMission<
                TSource, TOrthogonal>>
13          {
14          };
15          ...
16          class CbUSBLReceiveData : public cl_multirole_sensor::
                CbDefaultMultiRoleSensorBehavior<sm_turtle_hop::
                cl_usbl_receiver::ClUSBLReceiver>
17          {
18          public:
19              CbUSBLReceiveData()
20              {
21              }
22
23              virtual void onEntry() override
24              {
25                  ROS_INFO("CbUSBLReceiveData onEntry");
26                  this->requiresClient(sensor_);
27                  sensor_->onMessageReceived(&CbUSBLReceiveData::
                        onMessageCallback, this);
28                  sensor_->initialize();
29              }
30
31              template <typename TOrthogonal, typename TSourceObject
                    >
```

```
32              void onOrthogonalAllocation()
33              {
34                  postEventUSBLStartMission = [=]() {
35                      auto ev = new EvUSBLStartMission<TSourceObject
                            , TOrthogonal>();
36                      this->postEvent(ev);
37                  };
38                  ...
39              }
40
41              virtual void onMessageCallback(const ros_usbl_sensor::
                    USBLData &msg) override
42              {
43
44                  std::string data;
45
46                  data.assign(msg.data.begin(), msg.data.end());
47
48                  ROS_INFO("Received:\n%s", data.c_str());
49
50                  if (data == "Start Mission"){
51                      this->postEventUSBLStartMission();
52                  }
53                  ...
54              }
55              std::function<void()> postEventUSBLStartMission;
56              ...
57          };
58      } // namespace cl_usbl_receiver
59  } // namespace sm_turtle_hop
```

A.7 cl_control

A.7.1 cl_control.h

```
1   #pragma once
2
3   #include <smacc/client_bases/smacc_publisher_client.h>
4   #include <geometry_msgs/Twist.h>
5
6   namespace sm_turtle_hop
7   {
8       namespace cl_control
9       {
10          class ClControl : public smacc::client_bases::
                SmaccPublisherClient
```

```
11          {
12          public:
13              ClControl(std::string topicName) : smacc::client_bases
                    ::SmaccPublisherClient()
14              {
15                  this->configure<geometry_msgs::Twist>(topicName);
16              }
17          };
18      } // namespace cl_uuv_control
19  }
```

A.7.2 cb_control_heave.h

```
1   #pragma once
2
3   #include <smacc/smacc.h>
4   #include <smacc/smacc_client_behavior.h>
5   #include <sm_turtle_hop/clients/cl_control/cl_control.h>
6   #include <geometry_msgs/Twist.h>
7
8   namespace sm_turtle_hop
9   {
10      namespace cl_control
11      {
12          class CbVelHeave: public smacc::SmaccClientBehavior,
13                            public smacc::ISmaccUpdatable
14          {
15          private:
16              std::function<void()> deferedPublishFn;
17          public:
18              ClControl *publisherClient_;
19              float heaveSpeed_;
20              geometry_msgs::Twist cmd_vel_;
21
22              CbVelHeave(){
23              }
24              void setSpeed(float heaveSpeed) {
25                  ROS_INFO("Set Speed to %lf", heaveSpeed);
26                  heaveSpeed_ = heaveSpeed;
27                  cmd_vel_.linear.z = heaveSpeed_;
28                  this->setMessage(cmd_vel_);
29              }
30
31              template <typename TMessage>
32              void setMessage(const TMessage &data)
33              {
34                  deferedPublishFn = [=]() {
```

```
35                     publisherClient_->publish(data);
36                 };
37             }
38 
39             virtual void onEntry() override
40             {
41                 this->requiresClient(publisherClient_);
42             }
43 
44             virtual void update()
45             {
46                 if (deferedPublishFn != nullptr)
47                     deferedPublishFn();
48             }
49 
50             ...
51         };
52     } // namespace cl_control
53 } // namespace sm_turtle_hop
```

A.8 cl_uuv_control

A.8.1 cl_uuv_control.h

```
1  #pragma once
2  
3  #include <smacc/client_bases/smacc_action_client_base.h>
4  #include <sm_turtle_hop/UUVControlInitWaypointSetAction.h>
5  
6  namespace sm_turtle_hop
7  {
8      namespace cl_uuv_control
9      {
10         class ClUUVControl : public smacc::client_bases::
               SmaccActionClientBase<sm_turtle_hop::
               UUVControlInitWaypointSetAction>
11         {
12         public:
13             SMACC_ACTION_CLIENT_DEFINITION(sm_turtle_hop::
                   UUVControlInitWaypointSetAction);
14             ClUUVControl(std::string actionServerName);
15             virtual std::string getName() const override;
16             virtual ~ClUUVControl();
17         };
18     } // namespace cl_uuv_control
19 }
```

A.8.2 cl_uuv_control.cpp

```
1 #include <sm_turtle_hop/clients/cl_uuv_control/cl_uuv_control.h>
2
3 namespace sm_turtle_hop
4 {
5     namespace cl_uuv_control
6     {
7         ClUUVControl::ClUUVControl(std::string actionServerName) :
              Base(actionServerName)
8         {
9         }
10
11        std::string ClUUVControl::getName() const
12        {
13            return "TOOL ACTION CLIENT";
14        }
15
16        ClUUVControl::~ClUUVControl()
17        {
18        }
19    } // namespace cl_uuv_control
20 }
```

A.8.3 cb_uuv_control_check_waypoint_set_end.h

```
1 #pragma once
2
3 #include <smacc/smacc.h>
4 #include <sm_turtle_hop/clients/cl_uuv_control/components/
     waypoints_navigator/cp_waypoints_navigator.h>
5
6 namespace sm_turtle_hop
7 {
8     namespace cl_uuv_control
9     {
10        template <typename TSource, typename TOrthogonal>
11        struct EvWaypointSetEnd : sc::event<EvWaypointSetEnd<
              TSource, TOrthogonal>>
12        {
13        };
14
15        class CbCheckWaypointSetEnd: public smacc::
              SmaccClientBehavior
16        {
```

```
17         public:
18             WaypointNavigator *waypointNavigatorComponent_;
19             CbCheckWaypointSetEnd(){
20             }
21
22             virtual void onEntry() override
23             {
24                 this->requiresComponent(
                       waypointNavigatorComponent_);
25                 if(waypointNavigatorComponent_->
                       waypointSetComplete()){
26                     postEventWaypointSetEnd();
27                 }
28             }
29
30             template <typename TOrthogonal, typename TSourceObject
                   >
31             void onOrthogonalAllocation()
32             {
33                 postEventWaypointSetEnd = [=]() {
34                     auto ev = new EvWaypointSetEnd<TSourceObject,
                           TOrthogonal>();
35                     this->postEvent(ev);
36                 };
37             }
38
39             ...
40
41             std::function<void()> postEventWaypointSetEnd;
42         };
43     } // namespace cl_uuv_control
44 } // namespace sm_turtle_hop
```

A.8.4 cb_uuv_control_init_waypoint_set.h

```
1  #pragma once
2
3  #include <smacc/smacc.h>
4  #include <sm_turtle_hop/clients/cl_uuv_control/cl_uuv_control.h>
5  #include "uuv_control_msgs/WaypointSet.h"
6  #include <smacc/client_bases/smacc_action_client.h>
7  #include <sm_turtle_hop/UUVControlInitWaypointSetResult.h>
8
9  namespace sm_turtle_hop
10 {
11     namespace cl_uuv_control
12     {
```

```
13         class CbInitWaypointSet: public smacc::SmaccClientBehavior
14         {
15         public:
16             cl_uuv_control::ClUUVControl *uuvcontrolActionClient_;
17             std::vector<uuv_control_msgs::Waypoint> waypointset_;
18             CbInitWaypointSet(){
19             }
20 
21             virtual void onEntry() override
22             {
23                 this->requiresClient(uuvcontrolActionClient_);
24                 cl_uuv_control::ClUUVControl::Goal goal;
25                 goal.command = waypointset_;
26                 uuvcontrolActionClient_->sendGoal(goal);
27             }
28 
29             virtual void onExit() override
30             {
31                 //ROS_INFO("Entering ToolClientBehavior");
32             }
33         };
34     } // namespace cl_uuv_control
35 } // namespace sm_turtle_hop
```

A.8.5 uuv_control_action_server.cpp

```
1 #include <actionlib/server/simple_action_server.h>
2 ...
3 
4 typedef actionlib::SimpleActionServer<sm_turtle_hop::
    UUVControlInitWaypointSetAction> Server;
5 
6 class UUVControlInitWaypointSetActionServer {
7 public:
8     std::shared_ptr<Server> as_;
9 
10    bool trajectoryTracking;
11 
12    ros::ServiceClient controllerInitWaypoint_cli;
13    ros::ServiceClient hold_cli;
14    ros::Subscriber trajectory_tracking_on_sub;
15 
16    sm_turtle_hop::UUVControlInitWaypointSetGoal cmd;
17    sm_turtle_hop::UUVControlInitWaypointSetFeedback feedback_msg;
18 
19 
20    UUVControlInitWaypointSetActionServer() {
```

```
21          feedback_msg.result = sm_turtle_hop::
                UUVControlInitWaypointSetResult::STATE_IDLE;
22      }
23
24      void publishFeedback() {
25
26          as_->publishFeedback(feedback_msg);
27      }
28
29      void execute(const sm_turtle_hop::
            UUVControlInitWaypointSetGoalConstPtr &goal) {
30          cmd = *goal;
31          uuv_control_msgs::InitWaypointSet srv_init_waypoint_set;
32          uuv_control_msgs::Hold srv_hold;
33
34          // Configure waypoint message
35          srv_init_waypoint_set.request.heading_offset = 0;
36          srv_init_waypoint_set.request.interpolator.data = "lipb";
37          srv_init_waypoint_set.request.max_forward_speed = 0.4;
38          srv_init_waypoint_set.request.start_now = true;
39          srv_init_waypoint_set.request.waypoints = cmd.command;
40
41          controllerInitWaypoint_cli.call(srv_init_waypoint_set);
42
43          feedback_msg.result = sm_turtle_hop::
                UUVControlInitWaypointSetResult::STATE_IDLE;
44
45          // 10Hz internal loop
46          ros::Rate rate(10);
47
48          //wait for service to switch trajectoryTracking true
49          ROS_INFO("Waiting for trajectoryTracking");
50          while (!trajectoryTracking);
51          feedback_msg.result = sm_turtle_hop::
                UUVControlInitWaypointSetResult::STATE_TRACKING;
52          ROS_INFO("Entering loop");
53          while (ros::ok() && trajectoryTracking) {
54              if (as_->isPreemptRequested()) {
55                  //Stop current trajectory
56                  hold_cli.call(srv_hold);
57                  // a new request is being executed, we will stop
                        this one
58                  ROS_WARN("UUVControlInitWaypointSetActionServer
                        request preempted. Forgetting older request.");
59                  as_->setPreempted();
60                  return;
61              }
62              publishFeedback();
63              rate.sleep();
```

```
64          }
65          ROS_INFO("Finished loop");
66          feedback_msg.result = sm_turtle_hop::
                UUVControlInitWaypointSetResult::STATE_IDLE;
67          publishFeedback();
68          as_->setSucceeded();
69      }
70
71      void run() {
72          ros::NodeHandle nh;
73          ROS_INFO("Creating tool action server");
74          std::string controller_initwaypoint_name,
                controller_hold_name, trajectory_tracking_name;
75          std::string ns = ros::this_node::getNamespace();
76          controller_initwaypoint_name = "/" + ns + "/
                start_waypoint_list";
77          controller_hold_name = "/" + ns + "/hold_vehicle";
78          trajectory_tracking_name = "/" + ns + "/
                trajectory_tracking_on";
79          as_ = std::make_shared<Server>(nh, "
                UUVControl_action_server_node",
80                                        boost::bind(&
                                              UUVControlInitWaypointSetActionServer
                                              ::execute, this, _1),
                                              false);
81          ROS_INFO("Starting Tool Action Server");
82          hold_cli = nh.serviceClient<uuv_control_msgs::Hold>(
                controller_hold_name);
83          controllerInitWaypoint_cli = nh.serviceClient<
                uuv_control_msgs::InitWaypointSet>(
                controller_initwaypoint_name);
84          trajectory_tracking_on_sub = nh.subscribe<std_msgs::Bool>(
                trajectory_tracking_name, 1,
85                                                                  &
                                                                        UUVControlInitW
                                                                        ::
                                                                        trajectoryTrack
                                                                        ,

86                                                                  this
                                                                        )
                                                                        ;

87          as_->start();
88          ros::spin();
89      }
90
91      void trajectoryTrackingCB(const std_msgs::Bool::ConstPtr &msg)
             {
```

```
92          if (msg->data) {
93              trajectoryTracking = true;
94          } else {
95              trajectoryTracking = false;
96          }
97      }
98  };
99
100 int main(int argc, char** argv)
101 {
102     ros::init(argc, argv, "UUVControl_action_server_node");
103     UUVControlInitWaypointSetActionServer
            UUVControlInitWaypointSetActionServer;
104     UUVControlInitWaypointSetActionServer.run();
105     return 0;
106 }
```

A.8.6 cp_pose.h

```
1  #pragma once
2
3  #include <smacc/component.h>
4  #include <smacc/smacc_updatable.h>
5
6  #include <geometry_msgs/Pose.h>
7  #include <tf/transform_listener.h>
8  #include <tf/transform_datatypes.h>
9  #include <mutex>
10
11 namespace sm_turtle_hop{
12
13 namespace cl_uuv_control
14 {
15     class Pose : public smacc::ISmaccComponent, public smacc::
           ISmaccUpdatable
16     {
17     public:
18         Pose(std::string poseFrameName = "turtle3/base_link", std
               ::string referenceFrame = "world");
19         virtual void update() override;
20         void waitTransformUpdate(ros::Rate r = ros::Rate(20));
21
22         inline geometry_msgs::Pose toPoseMsg()
23         {
24             std::lock_guard<std::mutex> guard(m_mutex_);
25             return this->pose_.pose;
26         }
```

```
27
28          inline geometry_msgs::PoseStamped toPoseStampedMsg()
29          {
30              std::lock_guard<std::mutex> guard(m_mutex_);
31              return this->pose_;
32          }
33
34          inline const std::string &getReferenceFrame() const
35          {
36              return referenceFrame_;
37          }
38
39          inline const std::string &getFrameId() const
40          {
41              return poseFrameName_;
42          }
43
44          bool isInitialized;
45
46      private:
47          geometry_msgs::PoseStamped pose_;
48          static std::shared_ptr<tf::TransformListener> tfListener_;
49          static std::mutex listenerMutex_;
50          std::string poseFrameName_;
51          std::string referenceFrame_;
52          std::mutex m_mutex_;
53      };
54  } // namespace cl_uuv_control
55  }
```

A.8.7 cp_pose.cpp

```
1   #include <sm_turtle_hop/clients/cl_uuv_control/components/pose/
        cp_pose.h>
2   namespace sm_turtle_hop {
3       namespace cl_uuv_control {
4           std::shared_ptr<tf::TransformListener> Pose::tfListener_ =
                nullptr;
5           std::mutex Pose::listenerMutex_;
6           Pose::Pose(std::string targetFrame, std::string
                referenceFrame)
7                   : poseFrameName_(targetFrame),
8                     referenceFrame_(referenceFrame),
9                     isInitialized(false),
10                    m_mutex_() {
11              this->pose_.header.frame_id = referenceFrame_;
```

```
12         ROS_INFO("[Pose] Creating Pose tracker component to
               track %s in the reference frame %s",
13                  targetFrame.c_str(), referenceFrame.c_str());
14
15         {
16             //singleton
17             std::lock_guard<std::mutex> guard(listenerMutex_);
18             if (tfListener_ == nullptr)
19                 tfListener_ = std::make_shared<tf::
                       TransformListener>();
20         }
21     }
22
23     void Pose::waitTransformUpdate(ros::Rate r) {
24         bool found = false;
25         while (ros::ok() && !found) {
26             tf::StampedTransform transform;
27             try {
28                 {
29                     std::lock_guard<std::mutex> lock(
                           listenerMutex_);
30                     tfListener_->lookupTransform(
                           referenceFrame_, poseFrameName_,
31                                                          ros::Time(0),
                                                                transform
                                                              );
32                 }
33                 {
34                     std::lock_guard<std::mutex> guard(m_mutex_
                           );
35                     tf::poseTFToMsg(transform, this->pose_.
                           pose);
36                     this->pose_.header.stamp = transform.
                           stamp_;
37                     found = true;
38                     this->isInitialized = true;
39                 }
40             }
41             catch (tf::TransformException ex) {
42                 ROS_ERROR_STREAM_THROTTLE(1, "[Component pose]
                        (" << poseFrameName_ << "/[" <<
                       referenceFrame_
```

```
43                                                    << "] ) is failing on pose update : "
44                                                    << ex . what () ) ;
45                 }
46                 r.sleep();
47                 ros::spinOnce();
48             }
49         }
50 
51         void Pose::update() {
52             tf::StampedTransform transform;
53             try {
54                 {
55                     std::lock_guard<std::mutex> lock(
                           listenerMutex_);
56                     tfListener_->lookupTransform(referenceFrame_,
                           poseFrameName_,
57                                                  ros::Time(0),
                                                        transform);
58                 }
59                 {
60                     std::lock_guard<std::mutex> guard(m_mutex_);
61                     tf::poseTFToMsg(transform, this->pose_.pose);
```

```
62                     this->pose_.header.stamp = transform.stamp_;
63                     this->isInitialized = true;
64                 }
65             }
66             catch (tf::TransformException ex) {
67                 ROS_ERROR_STREAM_THROTTLE(1, "[Component pose] ("
                       << poseFrameName_ << "/[" << referenceFrame_
68                                                                     <<
                                                                          "]
                                                                          )
                                                                          is
                                                                          failing
                                                                          on
                                                                          pose
                                                                          update
                                                                          :
                                                                          "
                                                                          <<
                                                                          ex
                                                                          .
                                                                          what
                                                                          ()
                                                                          )
                                                                          ;
69             }
70         }
71     } // namespace cl_uuv_control
72 }
```

A.8.8 cp_waypoints_navigator.h

```
1 #pragma once
2 #include <smacc/smacc.h>
3 #include <geometry_msgs/Pose.h>
4 namespace sm_turtle_hop
```

```
5 {
6 namespace cl_uuv_control
7 {
8     class WaypointNavigator : public smacc::ISmaccComponent
9     {
10    public:
11        WaypointNavigator();
12        void loadWayPointsFromFile(std::string filepath);
13        const geometry_msgs::Pose getNextWaypoints();
14        bool waypointSetComplete();
15        int currentWaypoint_;
16    private:
17        std::vector<geometry_msgs::Pose> waypoints_;
18    };
19 } // namespace cl_uuv_control
20 }
```

A.8.9 cp_waypoints_navigator.cpp

```
1 #include <sm_turtle_hop/clients/cl_uuv_control/components/
     waypoints_navigator/cp_waypoints_navigator.h>
2 #include <fstream>
3 ...
4 namespace sm_turtle_hop
5 {
6 namespace cl_uuv_control
7 {
8 WaypointNavigator::WaypointNavigator()
9     : currentWaypoint_(0),
10      waypoints_(0)
11 {
12 }
13
14 bool WaypointNavigator::waypointSetComplete(){
15     ROS_WARN("waypoints_.size() %li - currentWaypoint_ %i",
          waypoints_.size(), currentWaypoint_);
16     if(currentWaypoint_>= waypoints_.size()){
17         return true;
18     }
19     else
20     {
21         return false;
22     }
23 }
24
25 const geometry_msgs::Pose WaypointNavigator::getNextWaypoints()
26 {
```

```
27   return waypoints_[currentWaypoint_++];
28 }
29
30 #define HAVE_NEW_YAMLCPP
31 void WaypointNavigator::loadWayPointsFromFile(std::string filepath
     )
32 {
33   this->waypoints_.clear();
34   std::ifstream ifs(filepath.c_str(), std::ifstream::in);
35   if (ifs.good() == false)
36   {
37     throw std::string("Waypoints file not found");
38   }
39
40   try
41   {
42
43 #ifdef HAVE_NEW_YAMLCPP
44     YAML::Node node = YAML::Load(ifs);
45 #else
46     YAML::Parser parser(ifs);
47     parser.GetNextDocument(node);
48 #endif
49
50 #ifdef HAVE_NEW_YAMLCPP
51     const YAML::Node &wp_node_tmp = node["waypoints"];
52     const YAML::Node *wp_node = wp_node_tmp ? &wp_node_tmp : NULL;
53 #else
54     const YAML::Node *wp_node = node.FindValue("waypoints");
55 #endif
56
57     if (wp_node != NULL)
58     {
59       for (unsigned int i = 0; i < wp_node->size(); ++i)
60       {
61         // Parse waypoint entries on YAML
62         geometry_msgs::Pose wp;
63
64         try
65         {
66           wp.position.x = (*wp_node)[i]["position"]["x"].as<double
               >();
67           wp.position.y = (*wp_node)[i]["position"]["y"].as<double
               >();
68           wp.position.z = (*wp_node)[i]["position"]["z"].as<double
               >();
69           wp.orientation.x = (*wp_node)[i]["orientation"]["x"].as<
               double>();
```

```
70            wp.orientation.y = (*wp_node)[i]["orientation"]["y"].as<
                  double>();
71            wp.orientation.z = (*wp_node)[i]["orientation"]["z"].as<
                  double>();
72            wp.orientation.w = (*wp_node)[i]["orientation"]["w"].as<
                  double>();
73
74            this->waypoints_.push_back(wp);
75          }
76          catch (...)
77          {
78            ROS_ERROR("parsing waypoint file, syntax error in point
                  %d", i);
79          }
80        }
81        ROS_INFO_STREAM("Parsed " << this->waypoints_.size() << "
              waypoints.");
82      }
83      else
84      {
85        ROS_WARN_STREAM("Couldn't find any waypoints in the provided
               yaml file.");
86      }
87    }
88    catch (const YAML::ParserException &ex)
89    {
90      ROS_ERROR_STREAM("Error loading the Waypoints YAML file.
            Incorrect syntax: " << ex.what());
91    }
92  }
93  } // namespace cl_uuv_control
94  }
```

A.9 cl_dvl

A.9.1 cl_dvl.h

A.9.2 cb_dvl_detect_floor.h

```
1  #pragma once
2  #include <sm_turtle_hop/clients/cl_dvl/cl_dvl.h>
```

```
3 #include <multirole_sensor_client/client_behaviors/
    cb_default_multirole_sensor_behavior.h>
4 namespace sm_turtle_hop
5 {
6     namespace cl_dvl
7     {
8         template <typename TSource, typename TOrthogonal>
9         struct EvDVLSeaFloorDetect : sc::event<EvDVLSeaFloorDetect
              <TSource, TOrthogonal>>
10        {
11        };
12        class CbDVLDetectFloor : public cl_multirole_sensor::
              CbDefaultMultiRoleSensorBehavior<sm_turtle_hop::cl_dvl
              ::ClDVL>
13        {
14        public:
15            double target_height_;
16            CbDVLDetectFloor(double height)
17            {
18                target_height_ = height;
19            }
20
21            virtual void onEntry() override
22            {
23                ROS_INFO("CbDVLDetectFloor onEntry");
24                this->requiresClient(sensor_);
25                sensor_->onMessageReceived(&CbDVLDetectFloor::
                      onMessageCallback, this);
26                sensor_->initialize();
27            }
28
29            template <typename TOrthogonal, typename TSourceObject
                  >
30            void onOrthogonalAllocation()
31            {
32                postEventDVLSeaFloorDetect = [=]() {
33                    auto ev = new EvDVLSeaFloorDetect<
                          TSourceObject, TOrthogonal>();
34                    this->postEvent(ev);
35                };
36            }
37
38            virtual void onMessageCallback(const
                  uuv_sensor_ros_plugins_msgs::DVL &msg) override
39            {
40                //Do other stuff with data if needed
41                if(msg.altitude != -1 && msg.altitude <
                      target_height_ + 0.5 && msg.altitude >
                      target_height_ - 0.5){
```

```
42                     postEventDVLSeaFloorDetect();
43                 }
44             }
45             std::function<void()> postEventDVLSeaFloorDetect;
46         };
47     } // namespace cl_dvl
48 } // namespace sm_turtle_hop
```

A.9.3 cb_dvl_detect_land.h

```
1  #pragma once
2  #include <sm_turtle_hop/clients/cl_dvl/cl_dvl.h>
3  #include <multirole_sensor_client/client_behaviors/
       cb_default_multirole_sensor_behavior.h>
4  #include <sm_turtle_hop/clients/cl_dvl/components/cp_dvl_sea_floor
       .h>
5  namespace sm_turtle_hop
6  {
7      namespace cl_dvl
8      {
9          template <typename TSource, typename TOrthogonal>
10         struct EvDVLSeaLandDetect : sc::event<EvDVLSeaLandDetect<
               TSource, TOrthogonal>>
11         {
12         };
13         class CbDVLDetectLand : public cl_multirole_sensor::
               CbDefaultMultiRoleSensorBehavior<sm_turtle_hop::cl_dvl
               ::ClDVL>
14         {
15         public:
16             CpDVLSeaFloor *dvlSeaFloorComponent;
17             ros::Time landTime;
18             bool landCheck;
19             CbDVLDetectLand()
20             {
21             }
22
23             virtual void onEntry() override
24             {
25                 ROS_INFO("CbDVLDetectLand onEntry");
26                 this->requiresClient(sensor_);
27                 sensor_->onMessageReceived(&CbDVLDetectLand::
                       onMessageCallback, this);
28                 sensor_->initialize();
29             }
30
```

```
31              template <typename TOrthogonal, typename TSourceObject>
32              void onOrthogonalAllocation()
33              {
34                  postEventDVLLandDetect = [=]() {
35                      auto ev = new EvDVLSeaLandDetect<TSourceObject, TOrthogonal>();
36                      this->postEvent(ev);
37                  };
38              }
39
40              virtual void onMessageCallback(const uuv_sensor_ros_plugins_msgs::DVL &msg) override
41              {
42                  this->requiresComponent(dvlSeaFloorComponent);
43                  auto lastDistance = dvlSeaFloorComponent->getSeaFloorDistance();
44                  //Do other stuff with data if needed
45                  if(msg.altitude == -1 && lastDistance < 2 && !landCheck)
46                  {
47                      landTime = ros::Time::now();
48                      landCheck = true;
49                  }
50                  if(ros::Time::now() - landTime > ros::Duration(1) && landCheck)
51                  {
52                      if(msg.altitude == -1 && lastDistance < 2)
53                      {
54                          postEventDVLLandDetect();
55                      }
56                      else
57                      {
58                          landCheck = false;
59                      }
60                  }
61              }
62              std::function<void()> postEventDVLLandDetect;
63          };
64      } // namespace cl_dvl
65  } // namespace sm_turtle_hop
```

A.9.4 cp_dvl_sea_floor.h

```
1  #pragma once
2  #include <smacc/component.h>
3  #include <smacc/smacc_updatable.h>
```

```
4  #include "uuv_sensor_ros_plugins_msgs/DVL.h"
5  #include <mutex>
6
7  namespace sm_turtle_hop
8  {
9      namespace cl_dvl
10     {
11         class CpDVLSeaFloor : public smacc::ISmaccComponent
12         {
13         public:
14             CpDVLSeaFloor(std::string dvlTopicName);
15             void dvlTrackCB(const uuv_sensor_ros_plugins_msgs::DVL
                   &msg);
16             inline double getSeaFloorDistance(){
17                 std::lock_guard<std::mutex> guard(m_mutex_);
18                 return this->seaFloorDistance;
19             }
20         private:
21             ros::Subscriber dvlSub_;
22             double seaFloorDistance = 0;
23             std::mutex m_mutex_;
24         };
25     } // namespace cl_dvl
26 }
```