Planning and Coordination of Multiple Autonomous Vehicles

Tiago Miguel Teixeira Sá Marques
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2016

Orientador
Luís Lopes, Professor, Faculdade de Ciências da Universidade do Porto

Coorientador
José Pinto & Paulo Dias, Investigador,
Faculdade de Engenharia da Universidade do Porto

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, / /

MSc

"To succeed, planning alone is insufficient. One must improvise as well"
- Isaac Asimov, Foundation

To my grandmother, Amélia

Acknowledgments

First of all, I would like to thank my parents and grandparents for the unconditional support and trust demonstrated throughout my academic life. Without them, every thing would've been much harder. To my sister Carla, for her patience in dealing with her grumpy brother.

I would also like to extend my gratitude to my advisors, José Pinto, Paulo Dias and Prof. Luís Lopes, for always being available to discuss ideas, provide feedback and material without which this project couldn't have been completed. To all the LSTS people for welcoming me into their team and showing interest in my work.

To the friends I made at Faculdade de Ciências da Universidade do Porto, that made this past years so much fun, in particular João Delgado for being so supportive, patience and providing rational thoughts during tougher times.

Finally, I want to thank Ana for her dedication and support that made me keep my head straight in the face of adversity. I could always count on you for being there when I most needed.

Agradecimentos

Gostaria de agradecer, acima de tudo, aos meus pais e avós pelo apoio e confiança incondicionais que demonstraram ao longo da minha vida académica. Sem vocês, teria sido tudo mais difícil. À minha irmã Carla pela paciência que tem em aturar o irmão rabugento.

Quero também estender os meus agradecientos aos meus orientadores, José Pinto, Paulo Dias e Prof. Luís Lopes, por estarem sempre disponíveis a discutir ideias, fornecer comentários e material sem o qual este projeto não teria sido concluído. A todas as pessoas do LSTS por me terem acolhido na equipa e demonstrado interesse no meu trabalho.

Aos amigos que fiz na Faculdade de Ciências da Universidade do Porto, por tornarem estes anos incríveis, em particular ao João Delgado pelo apoio, paciência e por me chamar à razão durante tempos mais difíceis.

Finalmente, quero agradecer à Ana pela dedicação e apoio que me fizeram manter motivado e confiante perante todas as adversidades. Pude contar sempre contigo quando mais precisei.

Abstract

An autonomous robot is one that performs tasks with a high degree of autonomy. Such robots have proven to be very prolific and important tools on a variety of scientific activities and even though, as a consequence, they are getting ubiquitous, their use is still very much reduced to a single vehicle per activity.

Heterogeneous networks of autonomous vehicles offer increased functionality through simpler interconnected robotic peers; instead of using a large monolithic vehicle, use simpler ones which also helps improving flexibility and robustness. However, in order to control and task such vehicular networks appropriately, one must be aware of the state of each system, their capabilities, current configurations, details of the area of operation, weather conditions, other vehicles operating in the area, *etc.* Furthermore, it must also account for changes in the environment, in mission and vehicles' requirements, as well as hardware failure.

LSTS has been designing, building and operating unmanned underwater, surface and air vehicle systems, to further develop *Networked Vehicle Systems* and operations. It has successfully operated networks of vehicles in various scenarios while also employing its software toolchain *Neptus-IMC-Dune*.

This dissertation proposes and describes an extension of LSTS's toolchain, for offline planning, and coordination of multiple autonomous vehicles, based on high-level objectives, automatic reasoning and mixed-initiative methods. This way I expect to reduce the overload on a human operator during the different stages of autonomous vehicles' operations. This framework, called *MvPlanning*, has been in missions of small scale, both in simulation (1 to 5 vehicles) and real-world scenarios (2 vehicles) such as in the *Recognized Environmental Picture* - REP16 exercise and in the networking and communication exercises of the *Networked Ocean* project, in *Slettvika* (Norway). The approach here presented aims to be a viable solution for planning and coordination of heterogeneous networks of vehicles, contrary to other approaches that focused on

the underwater version of such systems. It tries to maintain the operator aware of the coordination and planning steps without overwhelming it, but still require the its input so that he doesn't get over-reliant and confident or even unaware of the automated decisions being taken and why.

Contents

List of Acronyms

AIS	Automatic Identification System
APDL	Administração dos Portos do Douro, Leixões e Viana do Castelo
API	Application Program Interface
AUV	Autonomous Underwater Vehicle
AWT	Abstract Window Toolkit
BNF	Backus-Naur Form
CPP	Coverage Path Planning
EUROPA	Extensible Universal Remote Operations Architecture
GUI	Graphical User Interface
IMC	Inter Module Communication
IMU	Inertial Measurement Unit
JAXB	Java Architecture for XML Binding
LAUV	Light Autonomous Vehicle
LSTS	Laboratòrio de Sistemas e Tecnologias Subaquaticas
MER	Mars Exploration Rovers
MRA	Mission Review and Analysis
NVS	Networked Vehicle Systems
NVL	Networked Vehicles' Language
PDDL	Planning Domain Definition Language
REP	Rapid Picture Environment
RPM	Rotations Per Minute
SONAR	Sound Navigation and Ranging

STC	Spanning Tree Coverage
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
UI	User Interface
XML	eXtensible Markup Language

List of Figures

Chapter 1

Introduction

In this chapter I introduce the reader to what motivated this thesis 1.1 as well as the problem statement 1.3. Finally, I define the thesis' structure 1.4.

1.1 Motivation

An autonomous robot is one that performs tasks with a high degree of autonomy [1]. Some examples of autonomous robots are, *MER-Mars Exploration Rover* [2] - a car-size robotic rover that's exploring Mars - was built to investigate Martian climate and geology, planet habitability, *etc.* [2, 3], *Autonomous vacuum robots* whose purpose is to cover and vacuum an area efficiently, or *Unmanned Aerial Vehicles* tasked with fire detection on forests.

Planning of autonomous vehicles has been, typically, done by creating *a priori* a plan for them to execute, *i.e.* create a sequence of locations and parameters (velocity, altitude, *etc.*), that define the desired path for a vehicle to traverse. This kind of systems have proven to be very prolific and important tools on a variety of scientific endeavors, from space exploration to maritime data collection and surveillance, and even more so as technological advances reduce their cost and expand their autonomy, sensory capabilities and functionality. Even though, as a consequence, such vehicles are getting ubiquitous, their use is still very much reduced to a single vehicle per activity, even if more are available.

Heterogeneous networks of vehicles offer advantages such as decreased systems complexity; instead of using a large monolithic vehicle, use several simpler ones which also helps improving flexibility and robustness. However, in order to control and task such vehicular networks appropriately, one must be aware of the state of each system, their capabilities, current configurations, the state of the area of operation, weather conditions, other vehicles operating, *etc.* Such task becomes even more problematic when deploying systems whose tasks in the mission makes them operate out of communication range, for instance when an AUV goes underwater. Due to Human cognitive limitations [4], *e.g.* limited reaction speed, biased decision making, and the complexity of coordinating and planning multiple vehicles, it is critical to divide some of this work.

To avoid a single operator getting overwhelmed with decision making in these scenarios, more than one operators can be tasked with planning the missions, being each responsible for one assigned vehicle (or some other asset division). Still, this solution would be far from ideal, because besides n operators having to synchronize the plans and decisions between them, it's not cost-effective and practical to have an increasing number of operators and their resources (computer, *etc*), as the missions' complexity grows.

Traditional approaches on planning worked by giving each system an initial state and objective states and expecting them to create and execute the plan autonomously. However appealing, such levels of automation might not be desirable throughout all the systems' domains of application.

Some studies and experiments [5–7], suggest that removing the Human factor from path creation, leads to over-reliance on automated decisions, automation bias, complacency and loss of situation awareness, which, itself, might lead to lack of appropriate reaction from an operator, should something fail. Moreover, given the real-world dynamic environments in which such systems are deployed, planning is characterized by missing information, inaccurate models and changing objectives. It is not practical to model all possible real-world events and contingencies in a domain model and thus Human insight is required given its experience and judgment.

Considering the complexity of heterogeneous vehicular networks, the high variability of the vehicles' configurations, sensors and overall uncertainty, it seems very important to couple automation with Human judgment, hence using mixed-initiative methods in the planning process to offload work from the operator.

1.2 Networked Vehicle Systems

The *LSTS* - Laboratório de Sistemas e Tecnologia Subaquática (Underwater Systems and Technology Laboratory from Porto University) - has been designing, building and operating unmanned underwater, surface and air vehicle systems in order to build and further develop *Networked Vehicle Systems* (NVS) and operations. An NVS is a complex network composed of heterogeneous nodes (e.g. robots, sensors, routers, consoles). Nodes in the network have specific sensing, communication and moving capabilities and limitations which makes room for very interesting but complex system-level coordination and planning problems.

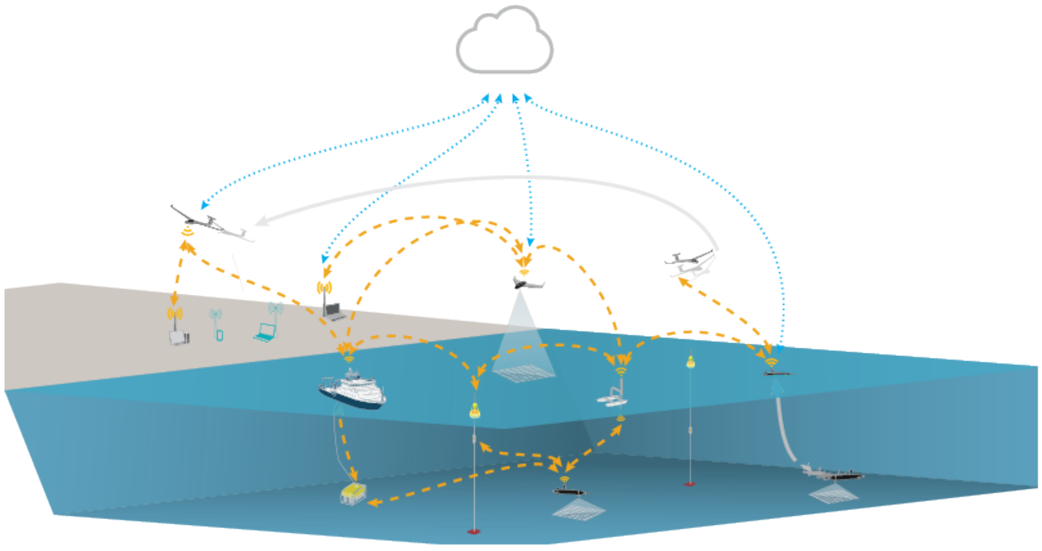


Figure 1.1: Networked vehicle systems

The LSTS was established in *1997* with researchers mainly drawn from Electrical and Computer Engineering, Mechanical Engineering and Computer Science backgrounds and since the early days the group has been focusing on the coordinated operation of vehicle networks.

During the last years, it has successfully utilized unmanned air, ground, surface and underwater vehicles in the Atlantic and Pacific oceans, and also in the Mediterranean sea, while also using its software toolchain *Neptus-IMC-Dune*. Furthermore *LSTS* has also been working with and testing *Delay Tolerant Network* protocols and architectures.

1.3 Problem Statement

LSTS has been developing software and hardware that allows the *simultaneous* control of, typically heterogeneous, multiple autonomous vehicles [8, 9]. Nonetheless, planning and coordinating such varied robotic platforms requires from the operator great amounts of attention and capacity of action, which inevitably takes its toll, leading to errors. This dissertation gathers concepts from previous LSTS experiments [10, 11], and with it I'm interested in building a framework that reduces the overload of Human operators while planning, coordinating and operating multiple vehicles, by creating an automated solution to help take complex decisions, all the while permitting the operator to supervise and intervene when and if necessary. It's main objectives are to extend *Neptus* by means of a plugin and:

- Extract capabilities from the available vehicles;
- Generate plans based on high-level definitions, such as "area to cover", "point to visit", *etc.*;
- Automatic allocation of the plans to the available vehicles, according to their capabilities an the plan's requirements;
- Allow multiple-vehicles to be controlled, simultaneously, towards common goals, set by an operator;
- Improve safety by imposing global restrictions and creating safeguards.

1.4 Dissertation Structure

This dissertation is organized as follows. Chapter 2 describes most of the research done, and background needed to understand this work, Chapter 3 illustrates in detail the software solution designed and developed to address the problem, called *MvPlanning*, Chapter 4 contains experiments in real-world scenarios designed to test the solution developed, as well as a discussion of its results. Finally, Chapter 5 contains a discussion of future work and developments to improve and expand *MvPlanning*'s capabilities.

Chapter 2

Background

This chapter will present the background and concepts necessary to understand the work developed and experiences conducted, that are described in later chapters. The software toolchain 2.1 that *MvPlanning* interfaces with and the systems 2.2 with which it was tested are described. A closer attention is given to the concept of planning 2.3

2.1 LSTS's Software toolchain

2.1.1 IMC - Inter Module Communication protocol

In this section the messaging protocol [12, 13] used between *LSTS*'s systems is described, as well as the most relevant messages for the work developed in this dissertation.

LSTS operates vehicles with different capabilities and target environments (ground, air and sea). Due to this heterogeneity, there's the need of a means of communication capable of abstracting the underlying configurations and hardware, allowing systems to interact seamlessly. The *Inter Module Communication Protocol* is a message-oriented protocol, developed at *LSTS*, understood by all vehicles and computer nodes in the network. It provides a shared set of messages, that can be serialized and transferred over different means. Its definition consists simply of an *XML* file where each message is defined.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<message id="263" name="Temperature" abbrev="Temperature">
  <description>
    Temperature measurement.
  </description>
  <field name="Value" abbrev="value" type="fp32_t" unit="Â?C">
    <description>
      Temperature value.
    </description>
  </field>
</message>
```

IMC messages are divided into categories, according to their purpose:

- **Navigation**: definition of the interface for stating a vehicle's navigation state.
- **Guidance**: define and report the parameters related to guidance in autonomous maneuvering. *E.g.*, heading, depth, velocity, etc.
- **Vehicle Supervision**: used to inform an external source about a vehicle's current state.
- **Plan Supervision**: define the specifications of a plan and it's life-cycle.
- **Sensors**: used to report sensor reading from hardware controllers. *E.g.*, LBL accoustic positioning system, GPS, IMU, etc.

2.1.1.1 PlanSpecification

One of the core *IMC* messages is the *PlanSpecification.* It is sent from a *Neptus* console to a vehicle and describes a plan and its parameters:

- **Plan ID**: The plan identifier
- **Starting Maneuver**: Identifier of the starting maneuver for the given plan
- **Maneuvers**: List of in-line messages of type *PlanManeuver*
- **Transitions**: List of in-line messages of type *PlanTransitions*

- **Start Actions**: (Optional) List of messages defining which actions to be executed on plan activation.
- **End Actions**: (Optional) List of messages defining which actions to be executed immediately to plan termination

  Prior to being sent to a vehicle, a *PlanSpecification* message is wrapped in a *PlanControl* message which informs the vehicle what to do with the plan, either to start, stop or load it (store in local database).

2.1.1.2 PlanManeuver

For the simpler "goto" behaviors - a maneuver intended to move a vehicle to a given location - an *IMC PlanManeuver* message describes a waypoint in a plan. It consists of an ID, a location (part of the "goto" behavior specification), and a list of *start* and *end* actions. The most common *start action* for a *PlanManeuver* is to switch on or off a payload (such as camera or sidescan sonar). For instance, in a plan to survey an area the vehicle needs to know when to start collecting data, instead of doing it as soon as the plan starts, wasting battery.

2.1.2 Neptus

Neptus [9] is a distributed *C3I(Command, Control, Communication and Information)* Java software for operations with autonomous and semi-autonomous networked vehicles, sensors and human operators. It allows operators to command and supervise LSTS's systems throughout a mission, by means of a visual interface (Operator Console) 2.1.

2.1.2.1 Neptus Console

A *console* 2.2 is the working environment of an operator inside *Neptus*. It is where all plans are created, and robots' information and telemetry displayed, allowing an operator to control and supervise missions. This feature is designed in a adaptable fashion which allows it to be customized and expanded through the means of plugins. Generally all new *Neptus* developments occur in the form of plugins.

Typical Neptus operator consoles consist mainly of a map, whose information can vary depending on the layers used (*Open Street Maps*, *S57* charts, etc), plugins to interact

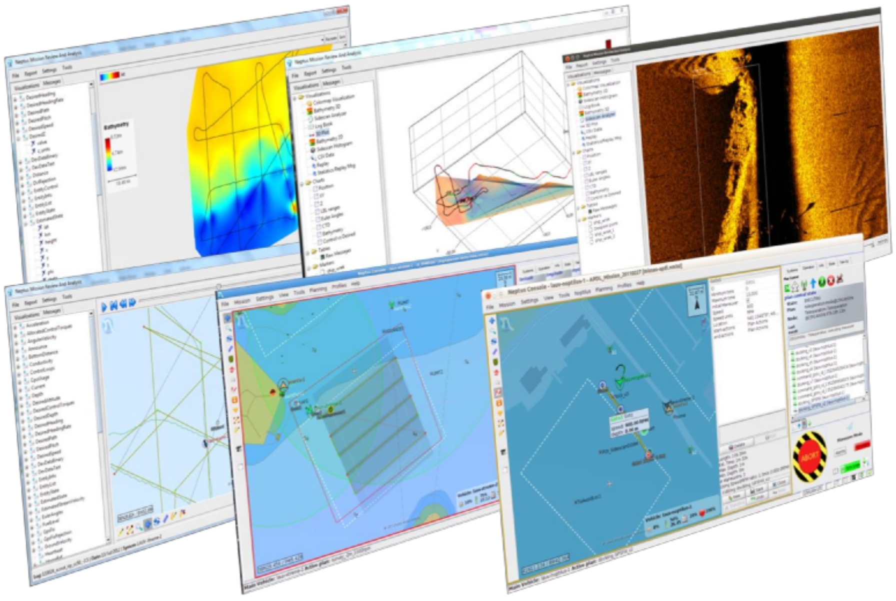


Figure 2.1: *Neptus*

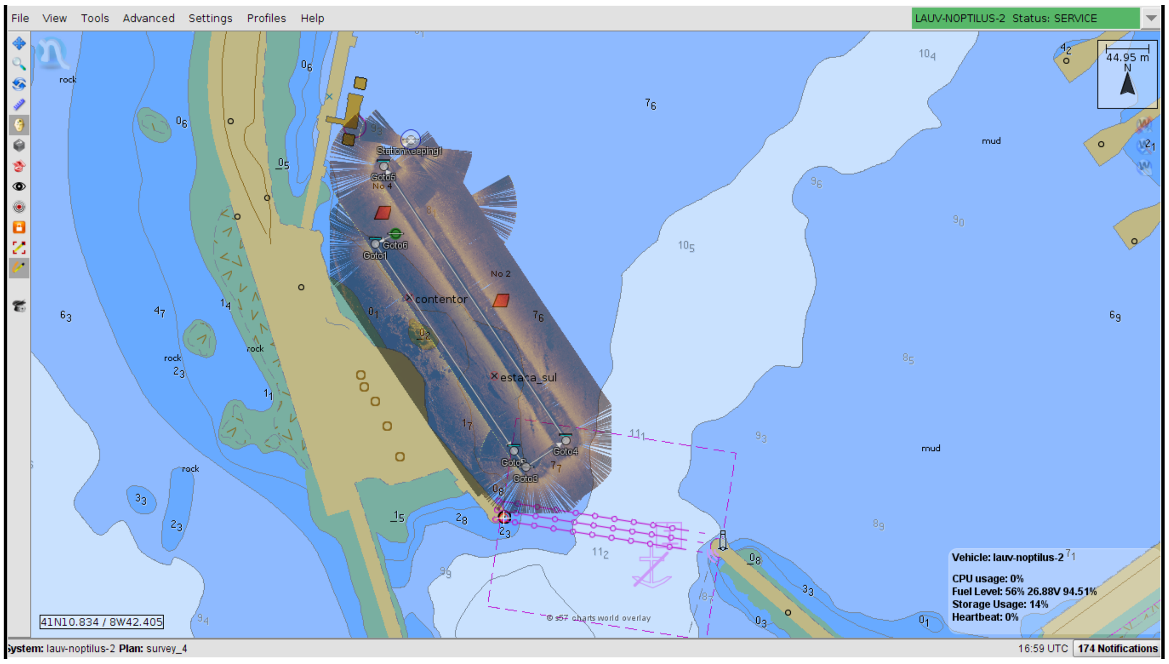


Figure 2.2: *Neptus* console

with it: *MapEditor* to add objects to the map, *PlanEditor* to create and/or edit plans, a *control panel* where plans can be chosen and sent to the vehicles, *etc.*, among other plugins that the operator might feel are necessary.

2.1.2.2 Mission Review and Analysis

One of the most important components of *Neptus* is the **MRA - Mission Review and Analysis** which provides support for the analysis of mission data such as sensors' data, executed plans' specifications and goals, exchanged messages, among others.

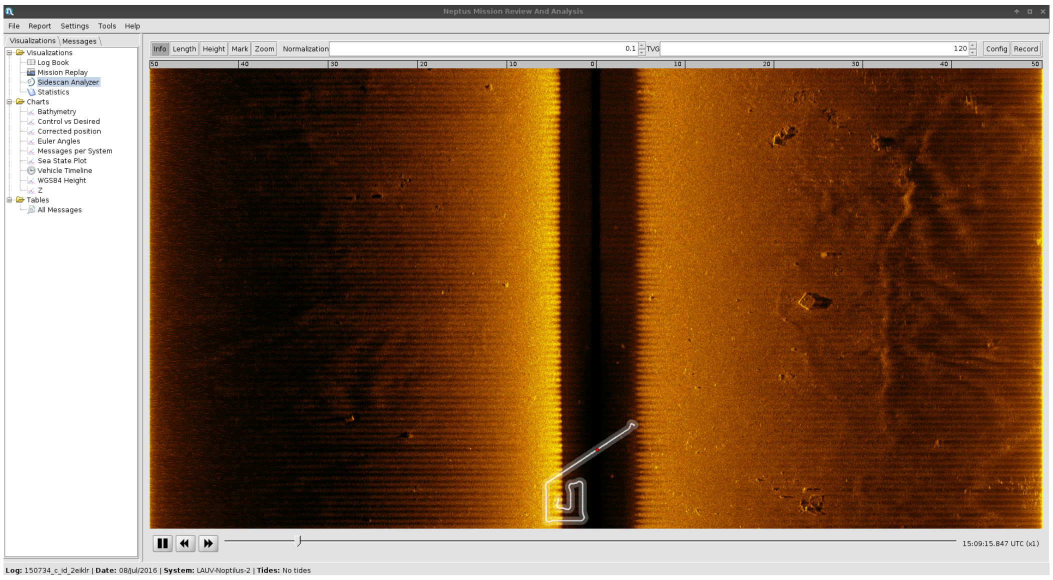


Figure 2.3: One of MRA's features, a sidescan analyzer

After the completion of a mission, the operator should gather its logs (both from *Neptus* and the vehicles) and, if necessary, analyze them through **MRA**.

2.1.3 DUNE

DUNE (DUNE Uniform Navigation Environment) is the on-board software that runs on all of *LSTS*'s vehicles and communication gateways. It is independent of CPU architectures and Operating Systems. It is modular and contains, just to name a few, modules for interaction with sensors and actuators, control, navigation and maneuvering. As with other nodes in the network, *DUNE*'s modules (also known as tasks) communicate using the *IMC* protocol.

2.1.4 Ripples

Ripples is a communications hub for data dissemination and situation awareness. It can be accessed through the web, using *REST* APIs or *Iridium*, and provides *real-*

time updates with asset positions, missions specifications and collected data. In sum, *Ripples* can be described as a Web/Iridium communications gateway than can also be used to for global situation awareness (as it stores all systems' positions and plans).

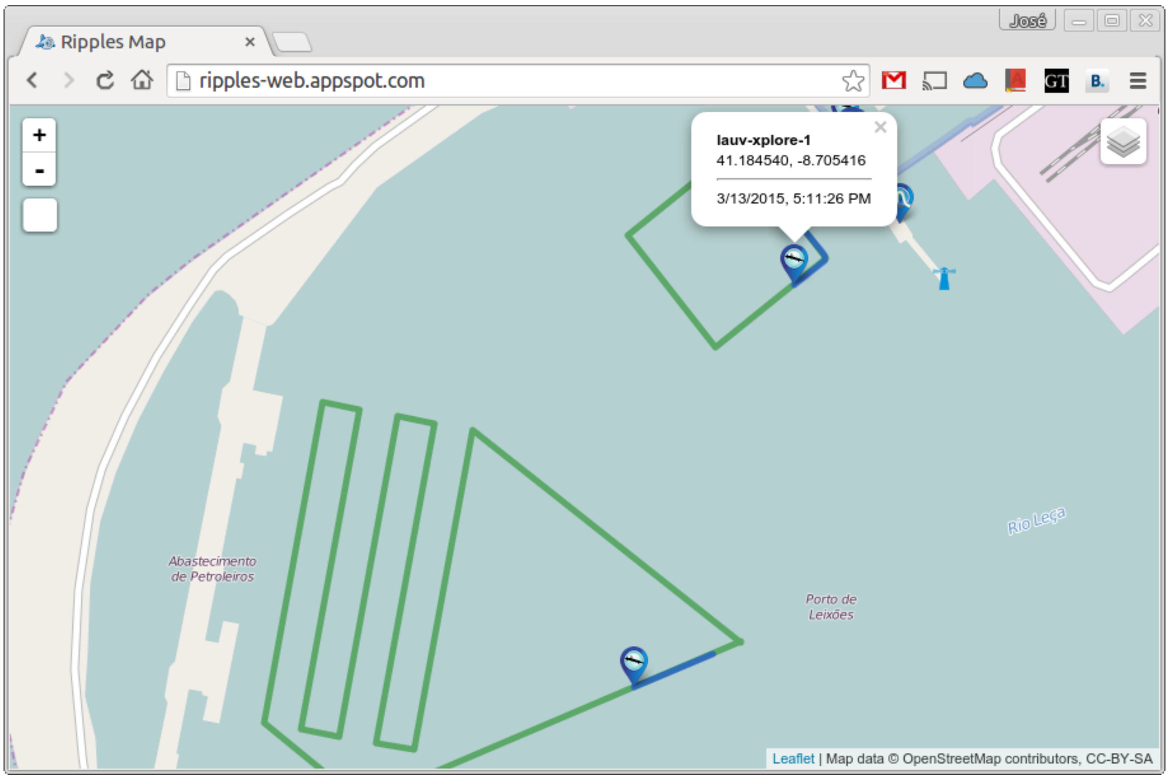


Figure 2.4: *Ripples*

Ripples allows an user, whether it is an operator or not, to visualize what is currently happening at different mission sites and also systems' information and position.

2.2 LSTS Systems

This section provides an overview of the hardware systems used by *LSTS*.

2.2.1 Vehicles

2.2.1.1 LAUV - Light Autonomous Underwater Vehicle

LAUV's development aimed at creating a low-cost and lightweight vehicle that could be easily launched, operated and recovered by a single operator.

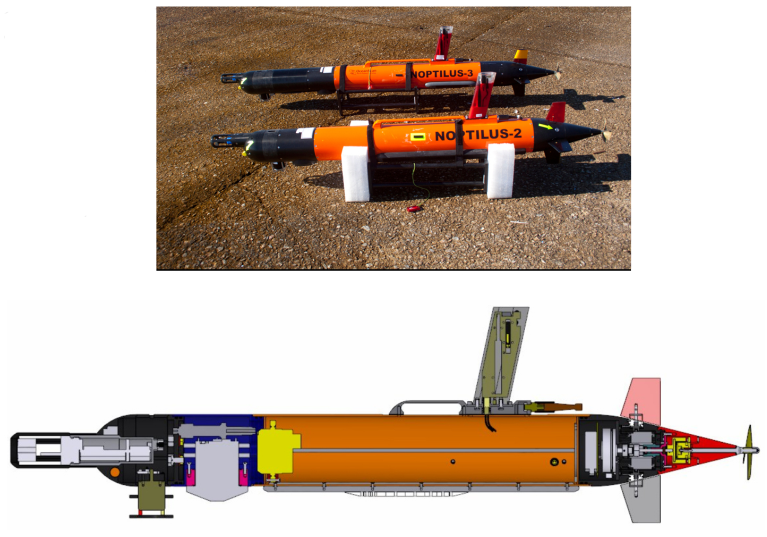


Figure 2.5: LAUV - Light Autonomous Underwater Vehicle

It's an affordable and highly effective surveying tool, built in a modular fashion, which allows it to be adapted to a variety of missions and scenarios, by adding to it sensors and payload modules, such as *Multibeam* and *Sidescan* sonars, *salinity* sensors and other environmental sensors (Crude and Refined Oils, and Rhodamine), among others.

2.2.1.2 X8 - Flying Wing

This low-cost COTS (Components Off-The-Shelf) vehicle is modified at *LSTS* and is aimed at surveillance missions and low altitude reconnaissance scenarios. It's able to provide live feeds from video camera and user other remote sensors. Due to its quick launch and recovery, it's perfect for fast algorithm testing, terrain mapping and operational surveillance. Besides the mentioned scenarios, it has been also used, by *LSTS*, for Wi-Fi coverage expansion missions.

2.2.2 Support Systems

2.2.2.1 Manta - Communications Gateway

This device (Figure 2.7) is used to create wireless and acoustic networks in the operations' areas, allowing the communication between the systems and operator.

It's also able to provide satellite links, and mobile Internet connection using 3G and

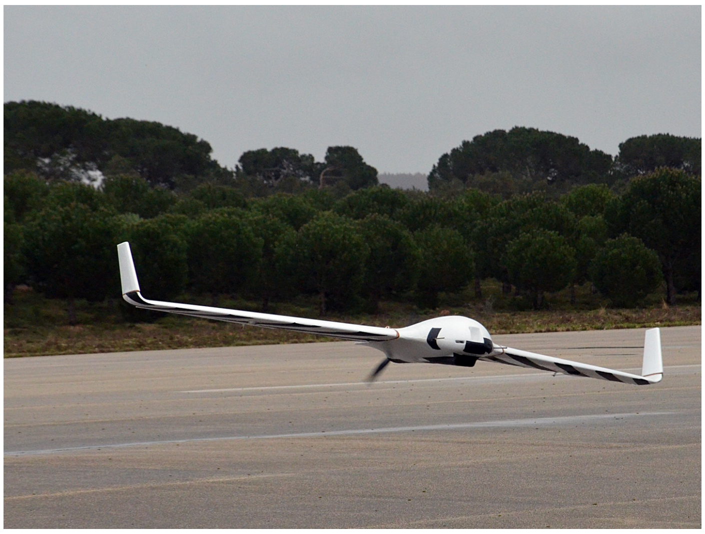


Figure 2.6: X8 Flying Win

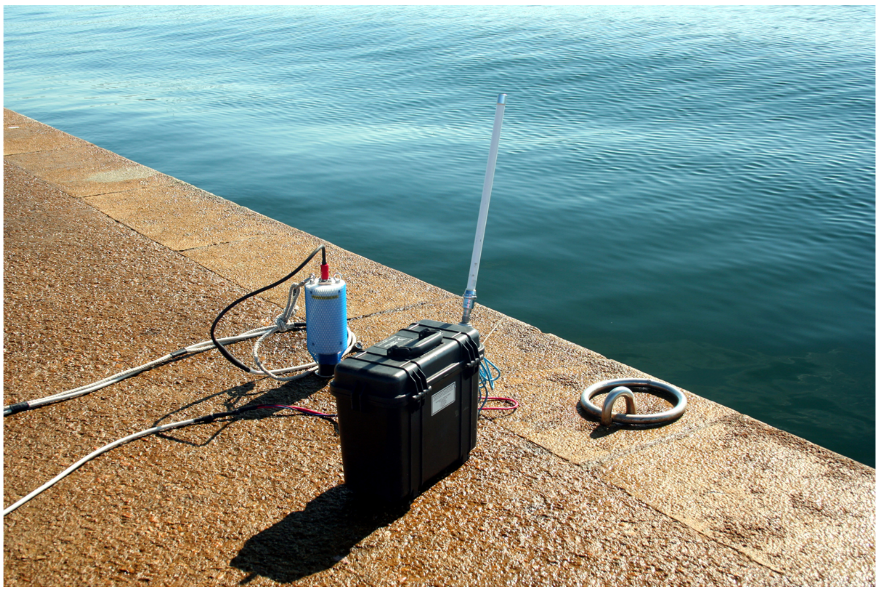


Figure 2.7: *Manta* communications gateway

4G networks. In some operational scenarios it's also deployed inside boats or on-board of autonomous vessels, this way creating a mobile network infrastructure.

2.3 Planning

Given a set of *initial* and *goal* states and possible *actions*, the objective of *Planning* is to find a sequence of *actions* one must take to achieve the *goal* states [14]. It involves the representation of both the actions and the world as models where *actions* are the set of tools an *agent*, *i.e.* the decision maker, can apply to the world in order to change its (the world's) state or its (the agent's) view of the world.

A **state space** is a pivotal aspect of any *Planning* problem. It models the different states the world can be in and is used by the *agent* when searching for a solution. It can be *discrete* if **finite** or **countably infinite**, or *continuous* if **uncountable infinite**. *E.g.* if a state represents a chess board after a piece was moved, then the **state space** represents all the permutations of moving the available chess pieces. The *state space* of a problem can easily grow beyond the level where a solution can be efficiently computed, and thus some heuristics might be needed to filter out the *non-optimal* states. The definition of *non-optimal* will be dependent on the *Planning* problem and the heuristics used. [14]

2.3.1 Planning definitions

Depending on the field of application, **Planning** might have different meanings. When studying *Artifical Intelligence* the *planning* problem might be to solve a *Rubik's cube*, the shortest path between point A and B, according to some heuristic, or any other task that can be modeled discretely. In *Robotics* the agent might be trying to solve the problem of deciding which path to take to reach a *goal*, or the best sequence of movements to achieve a desired robotic arm pose. Furthermore, in *Robotics*, there exist several types of problems one can solve with *Planning*. For instance, *Motion*, *Path* and *Trajectory* planning. This terms are often confused or used interchangeably in the literature. [14] It's also important to notice that most *planning* problems in *Robotics* are closely related or analogous to *Computer Science* ones such as *Path planning* and *shortest-path* computation.

- **Path planning** is concerned with finding, *geometrically*, the path to take from the initial to the final state, while avoiding possible obstacles;
- **Trajectory planning**, on the other hand, uses a path and represents it as a function of some variables, such as time, velocity, angle, etc. *I.e* where a robot needs to be in each instance of time and with a certain velocity;
- **Motion planning** is used to plan the actual *motion* of a robot having into account its kinematic and dynamic restrictions.

Because in the plan generation section of this thesis we're concerned with solving both types of problems, we'll use the term *Path planning* to mean both *Path* and *Trajectory* Planning. This is because when a final path is generated it will take into account both the path to take and its constraints/parameters.

2.3.2 Knowledge of the environment

Planners and planning algorithms can be classified according to their knowledge of the working environment, as:

- **Offline**: When the planning is done with *apriori* knowledge of the environment's state. This state is assumed to be static. These planners tend to be much simpler to implement but are more fragile to changes because the plan is generated against the current state of the environment and no longer changed.
- **Online**: There is limited or no apriori knowledge of the environment and thus the robot must use its sensors to acquire the environment's state (which is assumed to be dynamic, *i.e.* moving obstacles). In contrast with *offline* planners, these planners should account for changes in the environment and re-plan accordingly, which makes their implementation more complex but also more robust to dynamic environments.

2.4 Planning Languages

2.4.1 PDDL - Planning Domain Definition Language

PDDL is a standard encoding language for Artificial Intelligence planning, whose syntax is based on *Lisp* [15]. It is a simple standardization of a syntax to express semantics of actions, with pre and post-conditions that represent requirements and effects of actions. It's used both to describe the domain, **(1) domain description**, and planning problems, **(2) problem description**, within that domain, where **(1)** presents the components of the planning tasks, *i.e.* functions, predicates and actions, whereas **(2)** describes the specific problem to solve making use of the domain, *i.e* objects, initial conditions/state and goal-states. It can be used only to describe planning domains or in conjunction with a Planner to solve planning problems. *PDDL*'s notation is an Extended BNF (EBNF) where each rule is of the form

<syntactic element> ::= expansion

Furthermore, each file may only contain one PDDL definition of a domain or problem. The format of a PDDL domain is as follows

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
               (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
        ...)
  (:functions (FUNCTION_1_NAME ?A1 ?A2 ... ?AN)
               (FUNCTION_2_NAME ?A1 ?A2 ... ?AN)
        ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
   )
  (:action ACTION_2_NAME
    ...)
  ...)
```

Further work [16–18] has introduced *durative-action*, and a *:duration* element, allowing planners to tackle temporal planning. Following this development, in PDDL, plans are now interpreted as concurrent instead of sequential.

2.5 Coverage Area planning

2.5.1 Introduction

The *mobile robot covering problem* is defined as: Let a tool of a specific planar shape be attached to a mobile robot, and let A be a continuous planar world-area bounded by obstacles [19, 20]. Then the mobile robot has to move the tool along a path such that every point of A is covered by the tool along the path. [20–22] With this, we can define *Coverage Area planning* as the computation of a path that takes an agent to pass over all points of a given area or volume while avoiding any obstacle. This is a common scenario in Robotics with applications in *lawn mowers*, *vacuum cleaning robots* and *autonomous underwater vehicles*, among others. Even though these applications of *Coverage Area planning* are different they all share the same criteria: [20]

1. The robot must move through all the points in the target area, **covering it completely**
2. The whole region must be filled, by the robot, without overlapping paths
3. Continuous and sequential operation without any repetition of paths is required
4. All obstacles must be avoided
5. For simplicity in control, simple motion trajectories should be employed (*e.g.* straight lines or circles)
6. Aim for an "optimal" path.

Depending on the conditions and constraints, some of this criteria might be unattainable. In order to achieve coverage it is common, among *CPP* algorithms, to discretize the working space by decomposing it into sub-areas/cells.

2.5.2 Algorithms classification

Coverage Path Planning algorithms can be classified according to the decomposition applied to the working area. This is known as the *Choset's Taxonomy* [19]. Besides this classification, algorithms can also be classified according to their knowledge of the working environment, as **offline** or **online**.

They can be further classified as *Heuristic* or *Complete* depending on whether or not they *provably* guarantee complete coverage of a given area.

One alternative to finding the best path is to use, besides *heuristics*, *randomization*, *randomly select the next location of the path.* This technique is commonly employed in *vacuum cleaning robots* with good results, but for the scenarios where, for instance, *autonomous underwater vehicles* are used it might not be good enough due to battery and area size constraints. *E.g* 3D environments are too large for *randomization* to be useful and a vehicle might run out of battery before it covers the whole area. Also, besides not being a *complete* method, because of the abundance of overlapping paths that it usually generates, the quality of data might get compromised.

An algorithm to solve this problem tries to find the largest *Hamiltonian cycle* in the given area, *i.e* find the longest closed path which visits every node of the graph exactly once. The problem of finding an *Hamiltoninan cycle* for a general grid-like graph is known to be *NP-Complete*, but for certain grid-graphs, and for covering purposes, this can be achieved in O(N) by first constructing a finer-grained grid in *O(N)* time and then constructing an *Hamiltonian cycle* for this grid in additional *O(N)* time, for an N total number of cells. [19]

2.5.3 Choset's Taxonomy

As mentioned before, *CPP* algorithms can be classified according to the way they decompose the working space. Here, the ones reviewed fit into the class of **exact** and **approximate cellular decomposition**. Such algorithms break the working space into non-overlapping cells and their adjacencies are represented as a graph or matrix with cells being the nodes and their adjacencies, the edges or entries. Coverage plans based on this techniques often work in 2 steps:

1. Divide the working space into cells

2. Compute an exhaustive walk through the cells

Step 2 only gives the robot the sequence of nodes to traverse. Typically it has no information on how to actually do the coverage. Except for *Grid-based* methods, each cell needs then to be covered, before proceeding to the next one, with an explicit path.

2.5.3.1 Exact cellular decomposition

Exact cellular decomposition methods partition the free space into simple, smaller and non-overlapping regions (cells). Their union forms an exact representation of the free space.

2.5.3.2 Approximate cellular decomposition

Here, the free space and target area are decomposed into cells of the same size and shape, whose union is only an approximation of the original space. This is because it might not represent exactly the physical free space. Only a fraction of a given cell needs to be "obstructed" for it to be considered as an obstacle and thus "wasting" free space. Typically, when the robot enters a cell, it is considered as covered. Their size tends to be the same as that of the robot's footprint or sensor/effector's range. Complete coverage, if possible, is achieved by traversing every cell of the area.

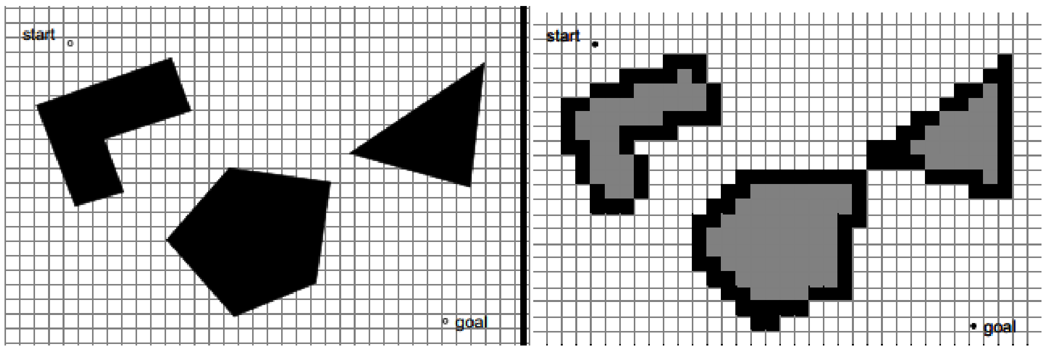


Figure 2.8: Example of an approximate decomposition.

2.5.4 Classical exact cellular decomposition

These are the foundational methods for *cellular decomposition*. Cells are generated by sweeping a line, from left to right, through the working space and every time something interacts with the line a new cell boundary is defined. For instance, when the line intersects an obstacle's vertice it marks the end of one cell and the beginning of the next one.

2.5.4.1 Trapezoidal decomposition

Trapezoidal decomposition is, perhaps, the simplest of the exact cellular decomposition methods. It generates a complete coverage path and each cell takes the form of a trapezoid. The exhaustive walk finds the path to traverse the cells and then the actual paths to traverse each cell are generated, typically using zigzag motion. Because this method relies on obstacle's vertices to define the cells, it only works in planar and polygonal obstacles and spaces.

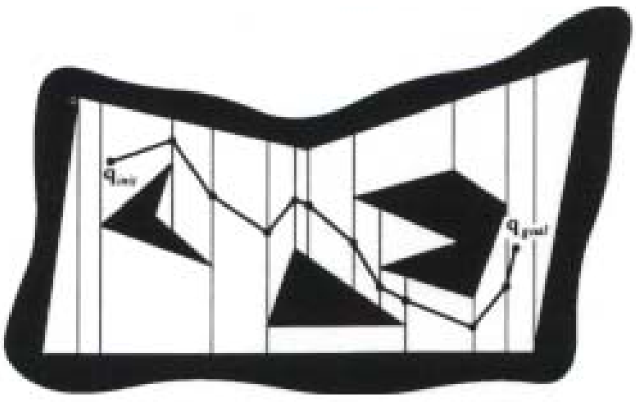


Figure 2.9: Exact cellular decomposition.

2.5.4.2 Boustrophedon decomposition

The way *Trapezoidal decomposition* decomposes the working space makes it generate several small cells that could be merged together to form bigger ones.

By creating a cell only when the sweep line encounters a vertex *and* it can be extended both above and bellow it (called *critical points*) the *Boustrophedon decomposition* reduces the number of cells and shorter paths are obtained. This method also assumes polygonal obstacles and *apriori* knowledge of the environment.

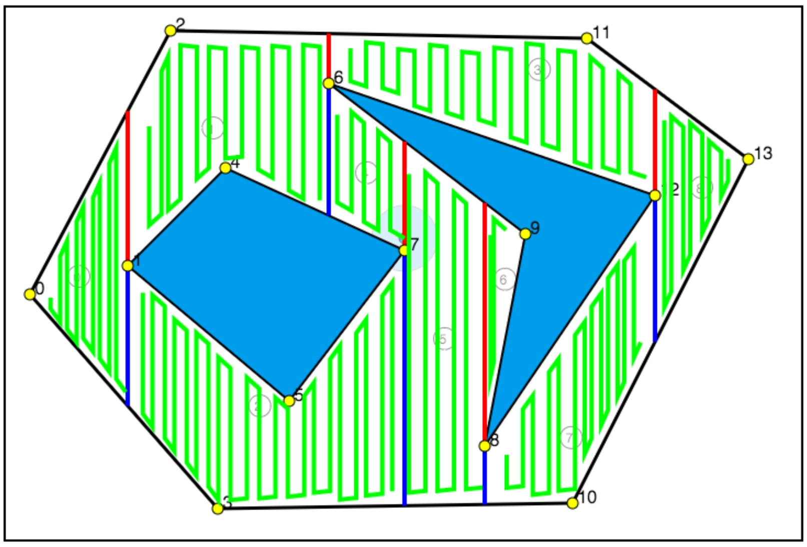


Figure 2.10: Boustrophedon decomposition and the coverage path generated.

2.5.5 Morse-Based cellular decomposition

Morse-Based cellular decomposition is a generalization of *boustrophedon decomposition* for non-polygonal areas and obstacles and can be used both *online* and *offline*. This method can also generate different coverage area patterns and cells shapes by changing the morse function that defines the sweep line. For the *boustrophedon decomposition* this morse function is $h(x, y) = x$.

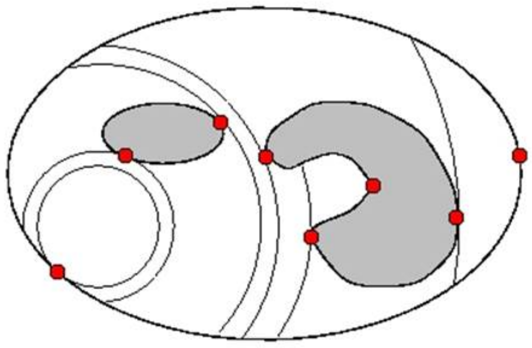


Figure 2.11: Morse-based cellular decomposition with morse function $h(x, y) = x^2+y^2$.

2.5.6 Grid-based cellular decomposition

Grid-based cellular decomposition methods divide the working area into uniform, and usually square cells. Each cell contains a "flag" that informs if it has an obstacle or not. This flag can be a boolean value (obstructed or not) or a number which states how probable it is that the given cell has an obstacle. According to Choset's taxonomy, *Grid-based decomposition* fits in the *approximate cellular decomposition* methods. As a consequence, this methods' completeness depend on the grid's resolution, *i.e.* "resolution-complete".

In figure 2.12, it's clear that instead of the vehicle going around/avoiding the obstacles' borders it uses an approximation formed by the cells that have an obstacle, and then a path is generated around them. This techniques are generally easy to implement but if a low resolution is used, one risks to leave some portions of the area uncovered. On the other hand, using a high resolution will vastly improve the amount of area that is covered, though, it is more memory-expensive and can cause some problems with the vehicles control and maneuverability. If there is a high number of small cells, due to the vehicles' dynamics it might not have the capability to maneuver through

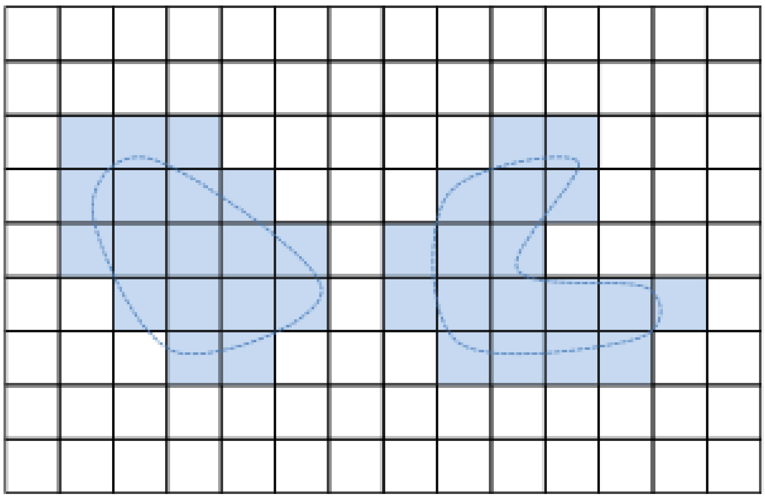


Figure 2.12: Grid-based cellular decomposition

them easily (difficult to do very fine movement adjustments). When choosing the cell's width, one should also have in mind the robot's task and tool used. The tool is what the robot uses to complete the task, *e.g.* a sidescan sonar. If planning for the typical vacuum cleaning robot, with omnidirectional locomotion and whose tool's range is short, the cell's can be, roughly, the size of the robot. On the other hand, when dealing with *autonomous underwater vehicles* that have less maneuverability and tools with greater range, the cells's size need to be much larger.

Some examples of such methods are *SpiralSTC*, *Hexagonal Grid* and *Neural Networks-based* coverage algorithms [20].

2.5.6.1 SpiralSTC algorithm

SpiralSTC, or *spiral spanning-tree coverage*, is a *grid-based* method, used both online and offline, that computes a *Minimum Spanning Tree* from the grid, to determine a high-level representation of the vehicle's path. Then, the actual path that the vehicle will traverse is computed, following the *spanning tree* edges. In the online approach the vehicle 'builds' the grid and *spanning tree* incrementally, using sensor information, while in the *offline* approach a "complete" tree and path are generated. Besides dividing the area into a grid, each cell (also called mega-cell) will be further divided into 4 smaller cells. The algorithm works as follows:

1. Divide the area into a grid

2. Divide each mega-cell into 4 smaller cells
3. Generate a *minimum spanning tree* where each node is a mega-cell
4. From a given start mega-cell compute a path that follows the spanning tree through the smaller cells, *i.e.* the vehicle moves along-side the *spanning tree* edges.

   Step *4* is crucial for the algorithm because it makes the vehicle move in a spiral and guarantees that each smaller cell is covered only once. This also has the advantage, over other *grid-based* methods, of not yielding situations where the vehicle moves to a "dead-end cell", where the it gets stuck and needs to go back through already visited nodes until it finds a new one, hence causing more overlapping paths and increasing the implementation's complexity.

The original authors [19] described 3 versions of the algorithm:

1. Offline SpiralSTC
2. Online SpiralSTC
3. Ant-Line STC

All three implementations need *O(N)* time to complete a covering path, however, both *1.* and *3.* need only *O(1)* additional memory for this task.

2.5.6.2 Offline SpiralSTC

This approach for coverage area tasks 3.1, assumes *a priori* and total knowledge of the working environment and thus a full path can be generated and sent to the robot, according to the environment's state.

Figure 2.13 shows an example of a decomposed area with the respective grid and, in red, its *Minimum Spanning Tree.* As already mentioned, in order to avoid obstacles (objects in yellow) the path/spanning tree completely avoids any obstructed cells, even if just a small portion of the cell is "obstructed" the path won't go near the obstacle.

After the working area is decomposed and a *spanning tree* is computed, one needs to generate the actual path to do the coverage. One example of such a path is figure 2.14 where everything computed by the algorithm is shown. The path in yellow (with

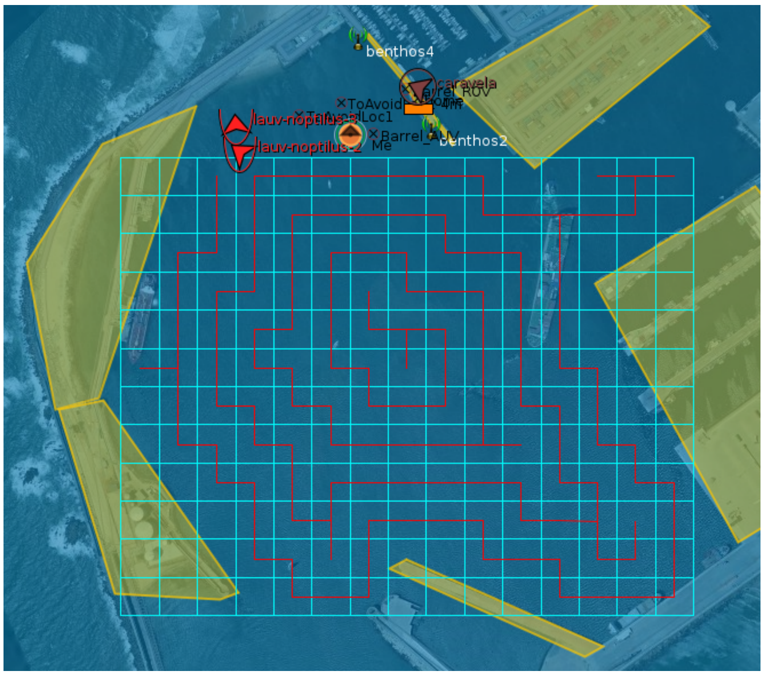


Figure 2.13: Grid decomposition and Minimum Spanning Tree,as implemented in *MvPlanning*

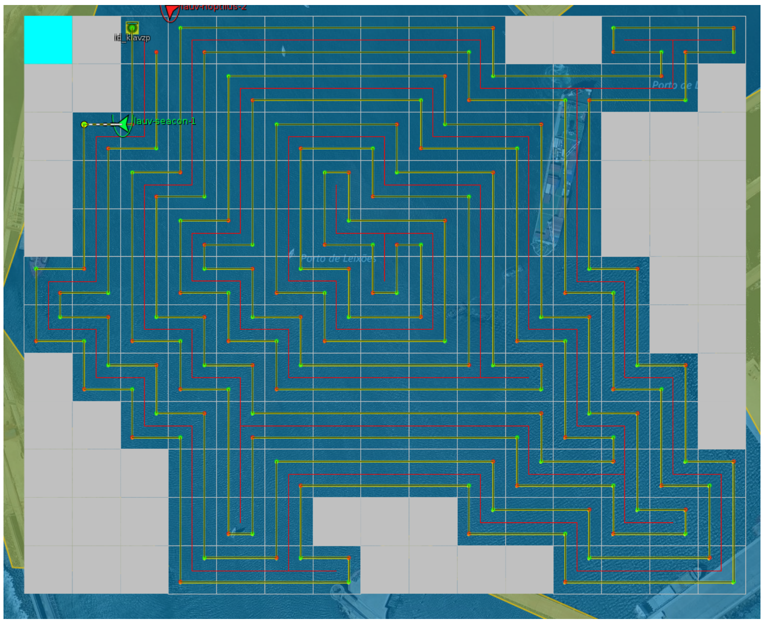


Figure 2.14: Path generated (yellow) around the spanning tree (red)

green and red waypoints) is the path the vehicle will traverse in order to go through every cell of the given area.

To generate the path that traverses the *spanning tree* through smaller cells, the algorithm starts at the first mega-cell and chooses, at each iteration, a new direction by choosing the first free mega-cell, from its neighbors, in an anti-clockwise fashion. This way, the vehicle will move on the right side (according to its direction) of the *spanning-tree* edges, creating a spiraled path, *i.e.* through the center locations of the sub-cells. After the completion of the plan, the vehicle will find itself back in the initial mega-cell, as desired.

Depending on the way the *spanning-tree* is computed one can influence the working-area's space that is covered by altering the weights of the edges between each cell. By giving, suitable, weights and using an algorithm such as Prim's [23], specific areas along a particular coordinate can be covered.

2.5.6.3 Online SpiralSTC

This version assumes no *apriori* knowledge of the working environment except that the obstacles are stationary. In order to traverse the working area, the robot needs to use sensory information to detect the obstacles and plan accordingly, and other sensors in order to know its position and orientation and also recognize the grid cells. The way the algorithm works is similar to the offline version, except that it works incrementally. The robot subdivides every cell it encounters, into four sub-cells. Then, it selects the next cell to move into, according to the rules specified previously, and a *spanning-tree* is constructed incrementally, until the whole area is covered. As the other versions, each neighbor of a given cell is scanned in a counter-clockwise fashion which makes the robot circumnavigate the *spanning-tree* also in the counter-clockwise direction. If the opposite direction was used, the path would also be generated in that direction.

Finally, contrary to the offline version, PRIM's algorithm cannot be used in the given problem, because it builds the *spanning-tree* by selecting *discontinuous edges*, while the robot needs information about the next physically continuous cell, to move into.

2.5.6.4 Ant-Line SpiralSTC

The Ant-Line SpiralSTC works the same way as the online version, but uses markers, *e.g*, color, odor, heat or other physical markers, to signal a visited sub-cell. By inspecting a sub-cell the robot can detect the marker and know that it has been visited already. The *O(1)* additional memory, contrary to online's *O(n)*, is a consequence of the behavior described previously. Using physical markers frees the robot from "having to remember" which cells have been visited. This can be confirmed by the robot, by sensing if all four sub-cells, of a given mega-cell, have markers. If this is the case, it means that the mega-cell has been completely covered and another one should be chosen, among the current neighbors.

2.6 Additional Concepts

2.6.1 Automatic Identification System

Automatic Identification System is a tracking system used on ships as means of identifying and locating them. It works by electronically exchanging data between nearby nodes, *i.e.* ships, *AIS* bases and satellites, through a standardized *VHF* transceiver. A ship broadcasting information through *AIS* informs other nodes about its *Navigation status*, *true heading and bearing*, *name*, *type of ship and cargo*, *dimensions* to the nearest meter, *draught* of the ship, among other important characteristics.

Neptus's plugin *ExternalSystemsHolder* contains information about *AIS* systems that are nearby, as long as an *AIS* transceiver is available to the operator and other vessels. To note that a vessel might have an *AIS* transceiver but not having it functioning. When an *AIS* system is detected, it's displayed in the *Neptus* console's map.

2.6.2 Ship Draught

A ship's *draught* (also known as *draft*) is the distance between the waterline and the bottom of the hull. In practice, it tells how much of the ship's hull is underwater. This is an important feature to know when operating *Unmanned underwater vehicles* in order to avoid damage to the systems.

2.6.3 Sonar - Sound Navigation and Ranging

The propagation of sound waves and their echo is the means by which a sonar works. It has applications in navigation, communication or detection of obstacles by, mostly maritime vehicles [24]. Typically the deployment of Sonar technology occurs underwater, where sound waves propagate faster than in air (at around 1.5 Km/s).

Contrary to most electromagnetic waves, acoustic waves are able to traverse long distances, underwater, without dispersing or losing too much energy to heat. Such characteristics makes sound waves perfect for underwater applications.

2.6.3.1 Sidescan Sonar

Sidescan is a type o sonar that transmits one beam of sound waves (pulse), per transducer, and analyzes the **energy** of the return signal (called echo) that bounces from the seafloor, fish, boats, *etc.* The roughness and hardness of the seafloor result in different energy variations of the return signal. With sidescan sonar it is possible to detect and estimate the materials, texture and morphology of the seafloor. A sidescan's transducer generally uses low-frequency signals, typically less than 20 kHz, transmitted in short pulses (< 2 ms). The greater the frequency the better resolution will be achieved but with the consequence of shorter range.

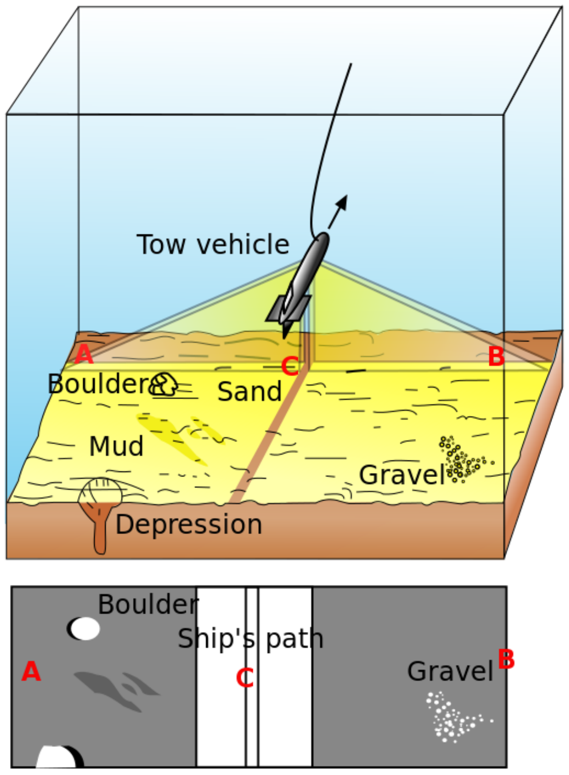


Figure 2.15: Sidescan sonar.
[1]Image adapted from `https://en.wikipedia.org/wiki/Side-scan_sonar`

In figure 2.15, the areas marked, in red, as **A** and **B** are areas where, typically, the return signal is weaker or has more noise, furthermore, the area **C** is called the sidescan's nadir. The nadir is a blind area right below the vehicle where there's absence or very low quality data (due to the way sound propagates from the transducers). By moving the sensor in a spiraled or row path it is possible to achieve data redundancy and better results on those areas. This technique is better described in figure 2.16:

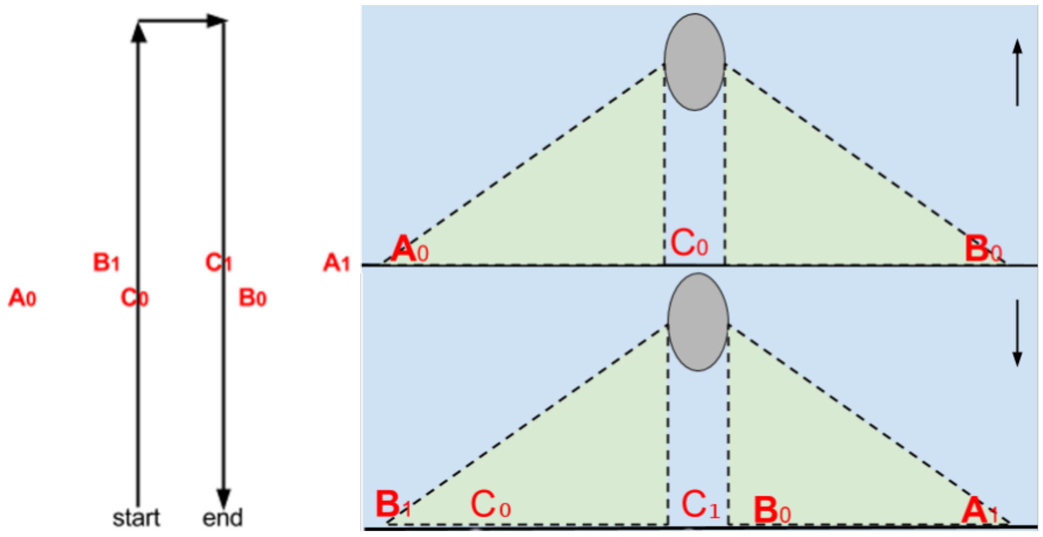


Figure 2.16: How to achieve data redundancy with sonar sensors

During the first pass, the sensor acquires data between A_0 and B_0, but not at C_0. On the next pass, in this case in the opposite direction, the vehicle's path is so that the sensor's beam still reaches some of the areas already surveyed in the previous pass. Consequently B_0 and C_0 will be correctly surveyed and any possible noise reduced.

2.6.3.2 Multibeam Sonar

Contrary to sidescan, multibeam sonar transmits several, narrower, beams (sometimes north of 240), and measures time differences between emission and reception, rather than energy variation. This way it can derive information about the seafloor's depth for each sound beam.

Chapter 3

Solution

The solution here described consists of a mixed-initiative coordination and high-level task-oriented planning framework such that the tasks to complete are added by the operator, forming a *planning problem* that is then solved by planning strategies. Its intention is to allow a single *Neptus* console (or any other type of console on the toolchain) and operator to easily setup a mission comprised of goals/tasks to be performed by a collection of heterogeneous vehicles connected and operated over radio and underwater communication networks.

This implementation aims to be a viable solution for planning and coordination of multiple heterogeneous vehicles , *i.e.* Underwater, Surface and Aerial autonomous vehicles, contrary to other approaches that focused on the underwater version of such systems. It tries to maintain the operator aware of the coordination and planning steps without overwhelming it, and balance the level of automation with the operator's input so that it doesn't get over-reliant and confident or even unaware of automated decisions. In sum, *MvPlanning*'s core notion is that operators can create or edit missions and plans by adding, editing, moving and deleting high-level objectives.

It's mainly composed of a module where the operator can add high-level objectives such as *survey location at X*, and obstacles, a generator to create plans 3.7.1, according to the defined objectives, and an allocator/planner that chooses which vehicles perform which plans. Furthermore, it has monitors and supervisors that allow both *MvPlanning* and the operator to be more aware of the planner and vehicles' states, *e.g. VehicleAwareness* and *Environment* modules pictured in Figure 3.1.

The software is built on top of *LSTS*'s toolchain 2.1 in the form of a *Neptus* plugin called *MvPlanning*, and using *IMC* to communicate with the systems, that are running

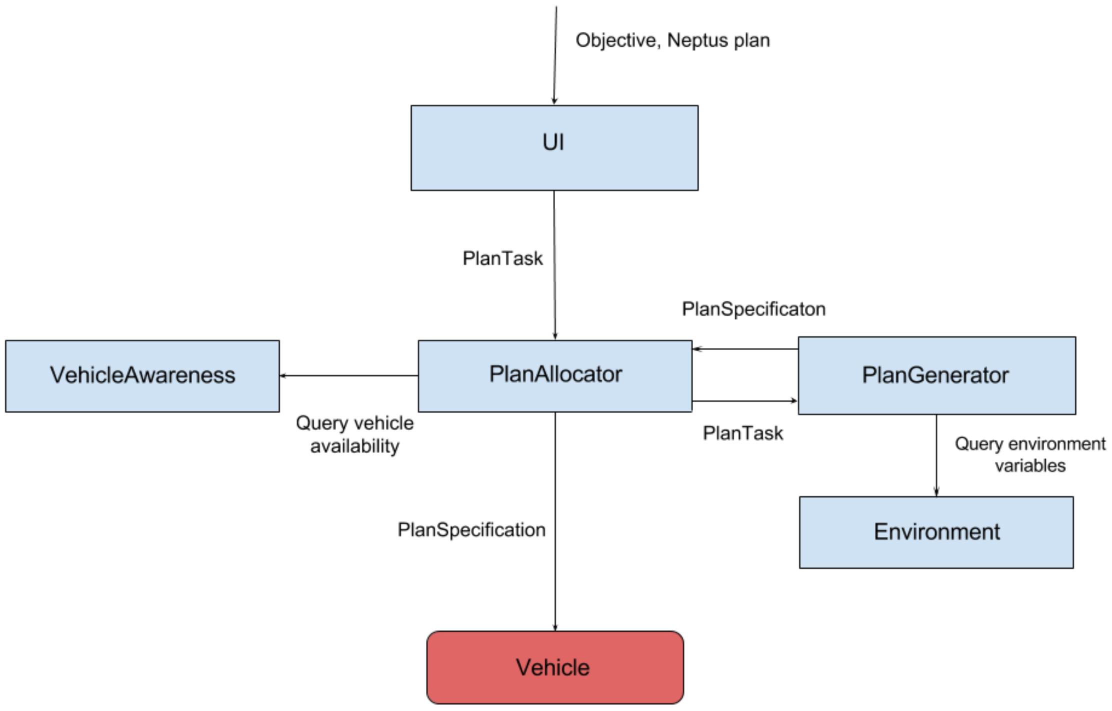


Figure 3.1: General architecture

the on-board software, *DUNE*. Its architecture (Figure 3.1) was designed "having in mind" the requirements defined at section 1.3. Sections 3.1 and 3.2 analyze some core *MvPlanning* concepts such as the definition of a Task.

3.1 Tasks

Tasks are the common denominator between *MvPlanning*'s modules. It is an abstraction of a plan 3.7.1 which can be seen as a sequence of maneuvers and its parameters (speed, altitude, *etc*).

When the user interacts with the *GUI* and adds a task, a new *PlanTask* object is created with all the information needed for generating the corresponding plan, and then it is sent for allocation, and so on until the actual plan is sent to the vehicle. A *PlanTask* consists mainly of an *id*, *profile*, *duration*, a *PlanType* object, a list of constraints and any additional data that other modules might need about the task.

Currently, the following tasks are supported:

- **VisitPoint**: Task used to move the vehicle to a given point, to survey it or perform any other job;
- **CoverageArea**: Given an area, try to cover it by passing through all its locations;
- **NeptusPlan**: Wrapper around plans made in Neptus (*PlanType*)
- **ToSafety**: Special task used, internally, to move vehicles in a safe manner, *i.e.* avoiding obstacles, between two locations.

  When creating a new task type, the abstract *PlanTask* and its methods need to be extended and implemented.

3.1.1 Task constraints

For a task to be allocated and completed successfully, some constraints have to be checked and validated. What these constraints are, depends on the task type. All tasks have predefined, and common constraints, but others can be created. All tasks share the following constraints:

- **IsAvailable**: Meaning that a vehicle is ready to start a new task
- **HasPayload**: If a given vehicle has the necessary payload, *i.e* sensors, to execute the task
- **HasTcpOn**: This task needs to be sent through a TCP connection, hence the vehicle must provide a TCP service for receiving IMC messages.
- **BatteryLevel**: The necessary value(s) of battery level needed for safe task completion. Currently the value(s) is/are predefined, but in the future, estimates of the vehicle's movement and payload's battery cost could be used to estimate how much battery a given vehicle will need to perform the task.
- **HasSafeLocationSet**: If a vehicle's safe location is set. The safe location is where the vehicle will return after finishing its assigned tasks. This is required so that safe paths can be computed.

The abstract class *TaskConstraint* can be extended by a sub-class in order to create a new constraint. The abstract method *isValidated()*, with the following prototype

public abstract <T> boolean isValidated(T... value);

must be implemented and used to validate the constraint, given any needed parameters.

In case the source used to load tasks constraints of a given task type fails, *PlanTask*'s method

public List<TaskConstraint> setDefaultTaskConstraints();

will be called, ensuring that a task type always has a set of constraints defined.

One way of defining and loading a task's constraints, besides the default method, is through a domain file containing a set of of predicates, objects and actions that describe a constraint. The following example is such a domain defined in *PDDL*:

```
(:durative-action CoverageArea
                  :parameters (?v - vehicle ?t - task)
                  :duration (= ?duration 15)
                  :condition (and (at start (IsAvailable ?v))
                                  (at start (IsActive ?v))
                                  (at start (HasPayload ?t ?v))
                                  (at start (HasTcpOn ?v))
                                  (at start (HasSafeLocationSet ?v))
                                  (at start (>= (BatteryLevel ?v) 60))
                                  )
                  )
```

3.1.2 Profile

Another important aspect of a task is its Profile. The profile has information about all the sensors, and corresponding parameters' values needed to perform the task, as well as the *speed* (in *meters per second* or *RPM*) and *Z* (depth or altitude) values.

It consists of a simple XML file and needs to be defined *a priori* by an operator in accordance to the mission and tasks' needs. What usually happens is that, for each vehicle there is a set of optimal payload profiles (e.g. sidescan low frequency, sidescan and for the most part, these parameters don't change much, which means that once a profile is created it won't likely need to be changed.

The XML definition is mapped to an actual *Profile* class through *JAXB* (Java API for XML Bindings). The following is an example of a profile for a Bathymetry task.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Profile Type="Bathymetry">
    <Z>3.0</Z>
    <Z_Units>DEPTH</Z_Units>
    <Speed>1.0</Speed>
    <Speed_Units>m/s</Speed_Units>
    <Payloads>
        <Payload type="Sidescan">
            <parameters>
                <parameter name="Active" value="True"/>
                <parameter name="Frequency" value="770"/>
                <parameter name="Range" value="30"/>
            </parameters>
        </Payload>
    </Payloads>
    <vehicles>
        <vehicle>lauv-noptilus-1</vehicle>
        <vehicle>lauv-noptilus-2</vehicle>
        <vehicle>lauv-noptilus-3</vehicle>
    </vehicles>
</Profile>
```

The profile above defines, in the root element *<vehicles>*, that *Noptilus* 1, 2 and 3 are supported, and a task should be performed with the *sidescan* sensor activated with a *frequency* of 770 and a *range* of 30 meters.

3.2 Operational area

When the planner starts, and before any task is added and allocated, an operational area needs to be computed. An operational area consists of a grid decomposition, with a predefined size, of the area of operation. It is centered in a mark with id of mvp_{oparea}. Though the high-level representation of this area (in the map) is a grid, its low-level one is of a grid-like graph where each grid cell is a node in the graph, and the edges represent boundaries between neighboring cells. Its size can be customized by an operator prior or after the area has been created.

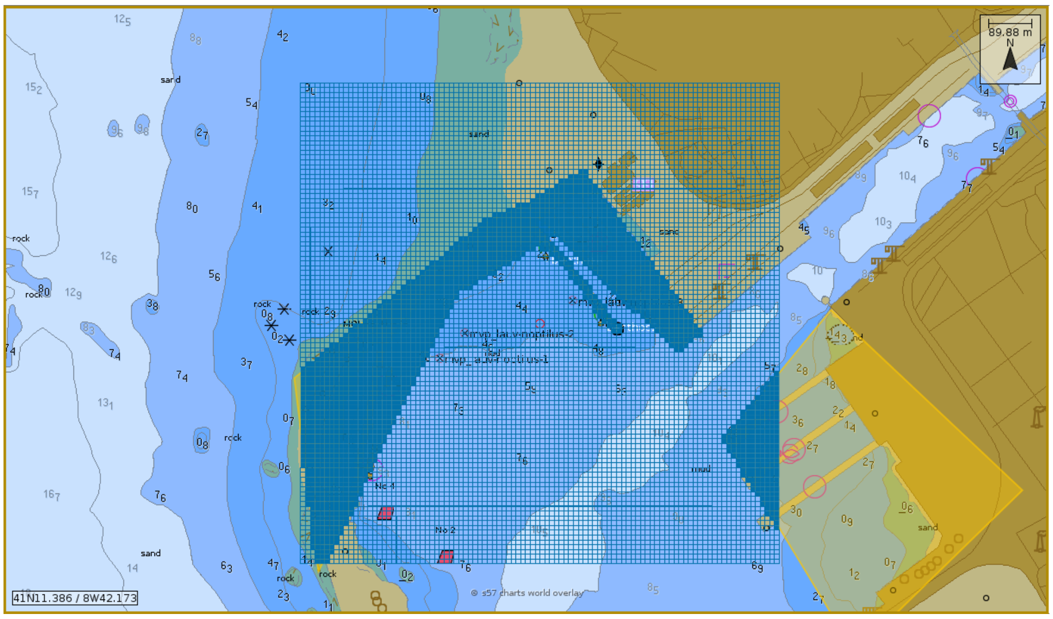


Figure 3.2: Operational area displayed on the map

It is used to compute safe paths between any two locations, allowing the vehicle to move safely, *i.e.* without colliding with obstacles, to and from the plans, thus representing the area of operation of the vehicles. Its grids' size shouldn't be neither too small so that the vehicle can't maneuver easily through it, nor too big so that a lot of free space will be wasted in case the area has obstacles (because the grid is an *approximate decomposition*). In case the area doesn't cover all the necessary locations, it has to be re-sized.

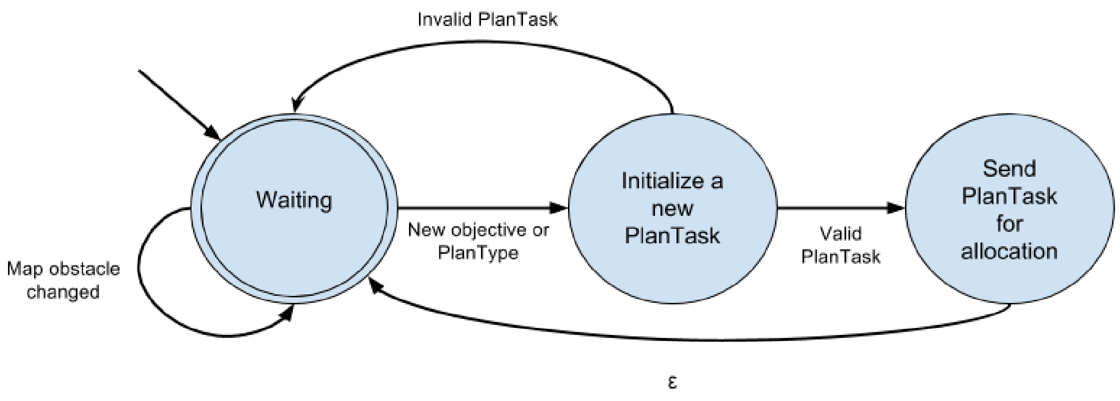

Invalid PlanTask
Waiting
New objective or PlanType
Initialize a new PlanTask
Valid PlanTask
Send PlanTask for allocation
Map obstacle changed
ε

Figure 3.3: User Interface's state machine

3.3 User interface

The user interface is responsible for receiving and displaying the tasks defined by the operator. Tasks are added to the console's map in the form of map objects, provided by Neptus' API through a plugin called *MapEditor*. There's also a window where tasks' name will be displayed and a button to send them for allocation. When doing so, the operator can choose which profile to associate to the task. To add *NeptusTask* tasks, the operator needs to select the plan from *Neptus*'s plans' tree (figure 3.4) and then click the *allocate* button on this window. *NeptusTask* task are handled differently because their plans are already generated. This window has other functionalities that are discussed on upcoming sections.

The objects added through *MapEditor*, when creating tasks, need to have their IDs prefixed by the string **mvp_**, *e.g.* **mvp_coverage_area** or **mvp_random_id**, so that *MvPlanning* knows exactly which map objects are meant to be tasks, and act accordingly. Otherwise, if not obstacles, they will be ignored. **ParallelepipedElement** objects (figure 3.11) are used to represent tasks of type *CoverageArea*, while *VisitPoint* tasks are added through *Mark* elements. Here the user can also add the vehicles' *safe locations*. After the completion of a task or if a vehicle's integrity is in danger, it will be moved to these locations.

Another important aspect of *MvPlanning*'s user interface are *Neptus*'s notifications, 3.5, *i.e* popup messages that appear in the console. This is used by *MvPlanning* to keep the operator in the loop by making him/her aware of the vehicle's state and which tasks are allocated to which vehicles.

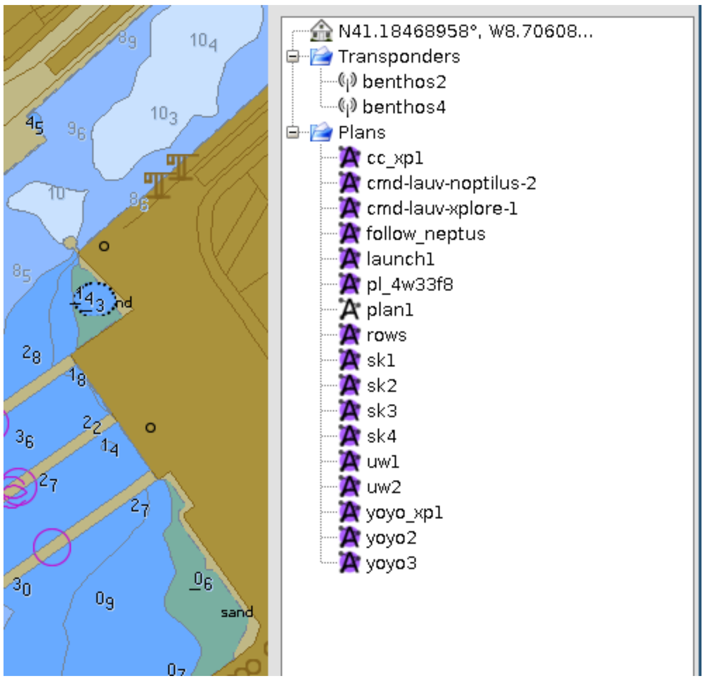
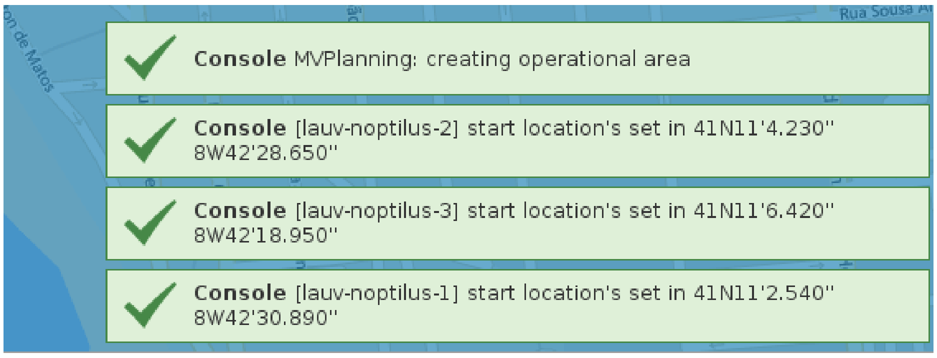


Figure 3.4: *Neptus*'s plans tree



Console MVPlanning: creating operational area

Console [lauv-noptilus-2] start location's set in 41N11'4.230" 8W42'28.650"

Console [lauv-noptilus-3] start location's set in 41N11'6.420" 8W42'18.950"

Console [lauv-noptilus-1] start location's set in 41N11'2.540" 8W42'30.890"

Figure 3.5: *Neptus*'s notifications

3.4 Monitors and Supervisors

Other important features of *MvPlanning* are the concepts of Monitor and Supervisor classes. A Monitor class is responsible for keeping track of certain variables and, possibly, having a public interface where other modules can query the variables' values. A Supervisor works also as a monitor but should take initiative of informing other modules of its readings, or even to take action if needed.

To give an example, consider a module A which might be responsible for monitoring the distance of a vehicle, currently doing a task, to an obstacle, *and* inform another module if this distance goes below a certain threshold. On the other hand, another module *B* might just be monitoring how much of a task has been completed at any moment in time. In order for a monitor class to be made a supervisor, it needs only to extend the class *AbstractSupervisor*.

Currently, supervisors can only interact with *PlanAllocator* and *PlanGenerator* (because those are the current needs of the application) but if other supervisors are added that need to interact with other modules, this can simply be done by adding a "reference" to it in *AbstractSupervisor* and create or modify constructors as needed.

3.4.1 VehicleAwareness

VehicleAwareness is the monitor responsible for maintaining and providing information about the current vehicles' state as well as use its information to validate task's constraints, when requested. During the life-cycle of a vehicle's operation, there are several operational states that it can be in. This states are attributed and provided only by *Neptus* and describe what the vehicle is currently doing:

- **Service**: The vehicle is connected, stationary and able to execute a plan;
- **Calibration**: Prior to the start of a plan, when the vehicle needs to turn on or calibrate sensors, etc;
- **Error**: The vehicle is in a state of error due to external or internal factors;
- **Maneuver**: When a plan is currently in execution;
- **Disconnected**: There's no connection to the vehicle;
- **Connected**: Currently connected but not yet able to execute a plan. Just prior to *Service* mode;
- **Boot**: Boot mode means that the vehicle is performing boot operations;
- **Finished**: When the vehicle is moving to the last waypoint of a plan or already arrived at the final location.

  For planning purposes, in *MvPlanning* 2 new states were created: *Available* and *Unavailable*. This states are only used internally and inform if the corresponding vehicle is apt to start a task/plan.

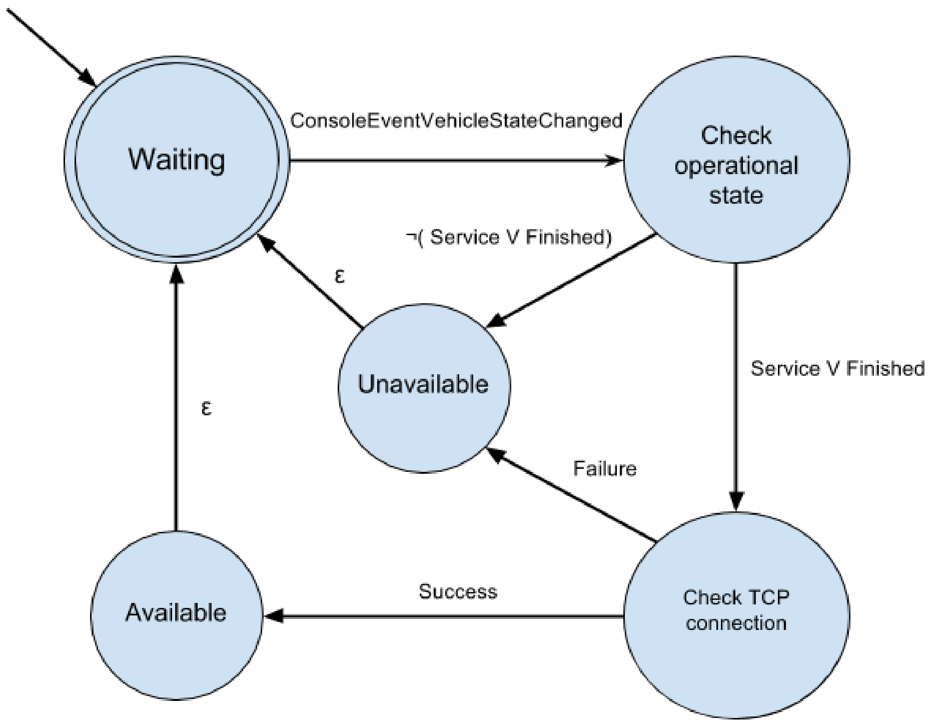


Figure 3.6: First VehicleAwareness version

Initially, for a vehicle to be considered as *Available*, only two metrics were used: an operational state of **Service** or **Finished**, and the possibility to establish a *TCP* connection to the vehicle. Every time a vehicle changed its operational state or a module queried a vehicle's availability, this metrics would be checked.

For more complex planning problems, a different approach is needed. For instance, the fact that a vehicle can communicate through *TCP* and is apt to start executing a task, doesn't say anything about the sensors it has and if they are suitable for the tasks created by the operator. Furthermore, it doesn't take into account vehicle's battery level, payload required, etc.

A new approach was used, where every task type has associated to it a list of constraints/pre-conditions (*TaskConstraint*) and the states *Available* and *Unavailable* are attributed only according to the vehicle's operational state. Then, when some module queries if a vehicle is available for a given task, *VehicleAwareness* evaluates all the constraints, and in case of success the task is allocated.

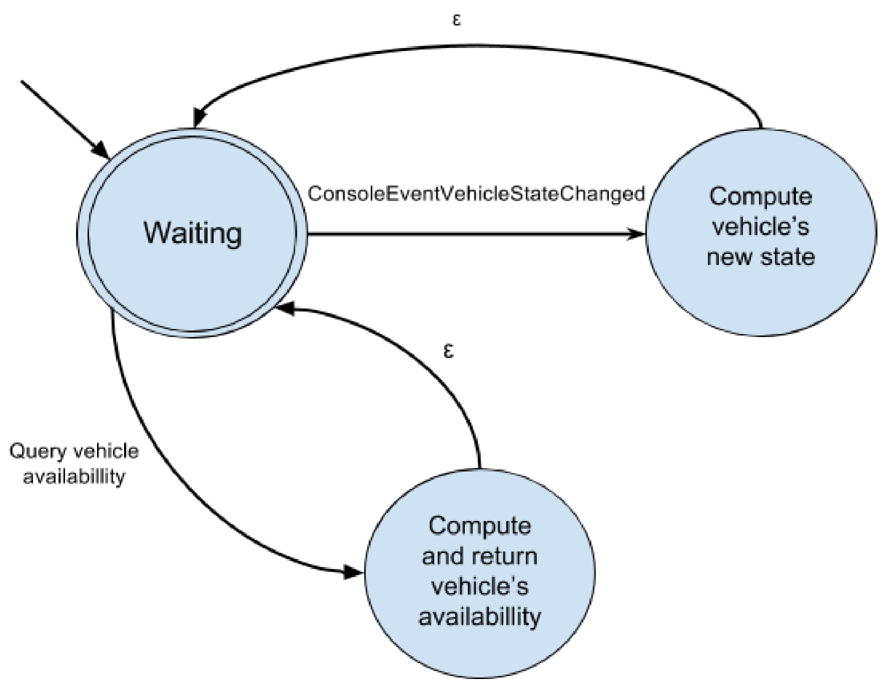

ε
Waiting
ConsoleEventVehicleStateChanged
Compute vehicle's new state
ε
Query vehicle availabillity
Compute and return vehicle's availabillity

Figure 3.7: Current version of VehicleAwareness

3.4.2 ExternalSystemsMonitor

ExternalSystemsMonitor is a supervisor responsible for monitoring external systems' (detected by Neptus's *ExternalSystemsHolder*) positions according to the vehicles, *and* inform *PlanAllocator* in case they get too close to one another.

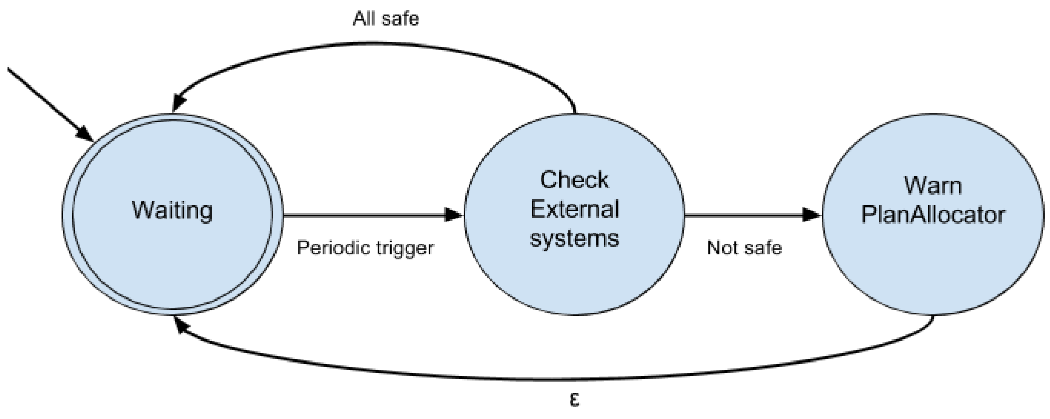


Figure 3.8: External systems monitor

Initially this module checked only if any of the known *external systems* was too close to any of the vehicles/systems in control by *MvPlanning*, but further optimization were done so that it could be more precise. For instance, even if an *external system* is too close to a vehicle, if its *draught* is not as deep as the vehicle's *Z*, it doesn't pose a problem. It's also not problematic, in the same distance scenario, if the *external system*'s bearing is not directed to the vehicle.

Because *external systems* are detected by means of an *AIS* system 2.6.1 this module is fallible, as it will only detect and warn about other ships provided that they are using such an identification system and that the captain has turned on the transponder (e.g. fishermen usually turn it off while fishing). If no ships in the area have their *AIS* system working, then *ExternalSystemsMonitor* won't be available to provide information.

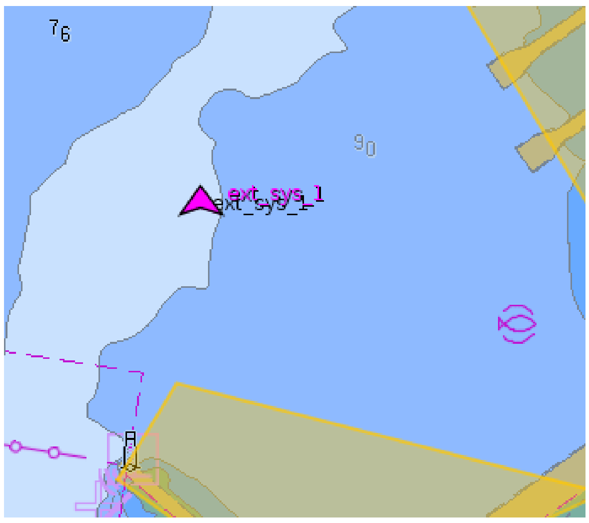


Figure 3.9: External system symbol in the map

3.4.3 Environment

This simple supervisor maintains a list of all the obstacles added to the map and provides a set of methods that can be used to check if given map objects/elements (*AbstractElement*) or an area (Java AWT's Area) intersect with any obstacle.

This module is subscribed to *Neptus*'s events bus and listens to *MapChangeEvent* events to know everytime an object was added, removed or simply changed in the console's map. Every time this object is an obstacle, then the list gets updated and *PlanGeneration* is asked to update the operational area accordingly.

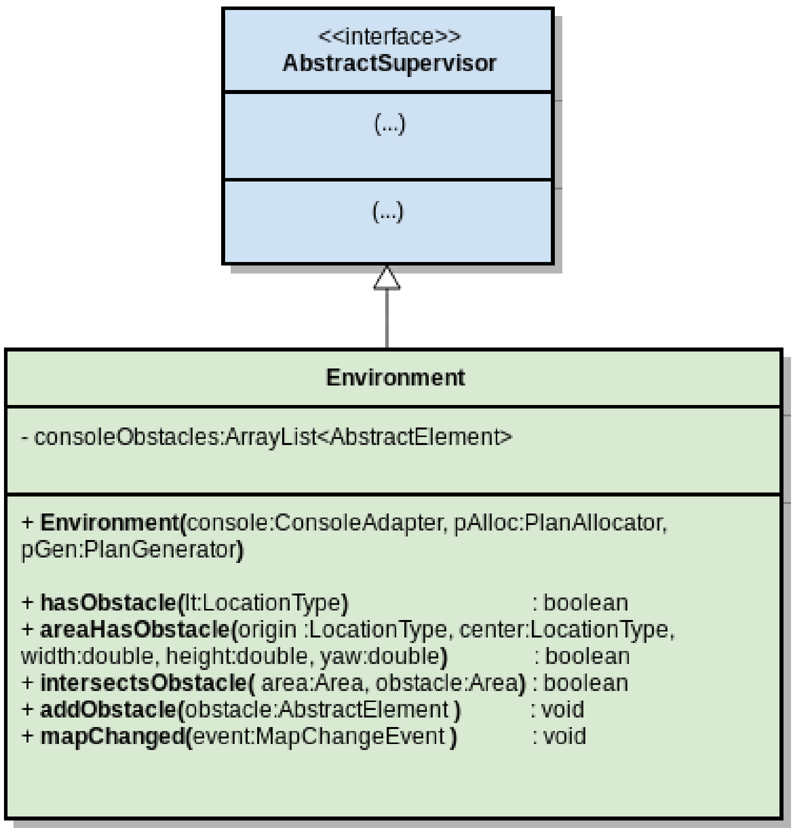

<<interface>>
AbstractSupervisor
(...)
(...)
Environment
- consoleObstacles:ArrayList<AbstractElement>
+ Environment(console:ConsoleAdapter, pAlloc:PlanAllocator, pGen:PlanGenerator)
+ hasObstacle(lt:LocationType) : boolean
+ areaHasObstacle(origin :LocationType, center:LocationType, width:double, height:double, yaw:double) : boolean
+ intersectsObstacle(area:Area, obstacle:Area) : boolean
+ addObstacle(obstacle:AbstractElement) : void
+ mapChanged(event:MapChangeEvent) : void

Figure 3.10: UML diagram of *Environment*

3.4.4 StateMonitor

StateMonitor informs the operator, through the *GUI*, about the current state of the application, or when it gets changed:

- **Running**: When tasks, if they exist, are currently being allocated;
- **Paused**: State triggered by the user, to stop momentarily task allocation
- **Waiting**: State used only by *MvPlanning*, when some computations or configurations need to be carried out before task allocation is possible.

The application starts in *waiting* state, and transitions to *paused* only after the *operational area* is computed. This is because task allocation is not possible without the computation of safe paths. As soon as the operator is ready, it can start *MvPlanning* by changing its state to *running*. At any time the state can be put into *Paused*, halting any future allocations. In this state any monitor and supervisor will still be working, maintaining security and consistency of the systems and environment's state. The reason that *MvPlanning* goes from *waiting* to *paused* instead of *running*, is that otherwise vehicles would start performing tasks right away, and the operator might not be paying attention or be ready to supervise the mission.

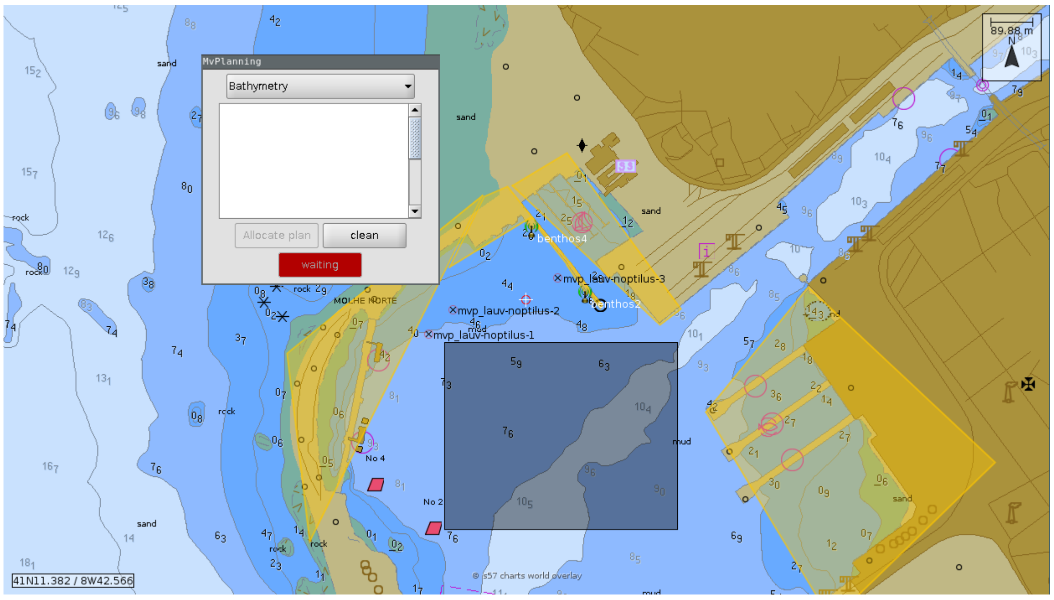


Figure 3.11: Neptus and *MvPlanning UI*, and *Waiting* state. The black box is a **ParallelepipedElement** provided by Neptus, to draw areas in the map

In figure 3.11 it's possible to see that there's no operational area yet, and *MvPlanning*'s state is *Waiting*. Also depicted is a coverage area added by an operator, in the map, soon to be generated and allocated if any vehicle is available, and when the state changes to *Running*.

3.5 Plan generation

PlanGenerator's role, Figure 3.12, is to receive a *PlanTask* object, generate the corresponding plan and complete the task by adding to it the generated *PlanType*. In other words, *PlanGenerator* maps high-level objective to a sequence of locations and parameters, so that the target vehicles can understand and execute them. Depending on the task type and its characteristics, the generated plan might be divided in several smaller ones, resulting that more *PlanTask* objects are created. For instance, the generated plans for coverage area tasks tend to have long completion times, sometimes more than one or two hours. If it is expectable that more than one vehicle will be available and able to execute the plan, it makes sense to divide it, for example, in smaller plans of 15 minutes. Another reason to divide a plan is if it's completion time is bigger than a pre-defined value by an operator. After the *PlanType* is divided, the resulting ones are added to new *PlanTask* objects and passed on to *PlanAllocator*.

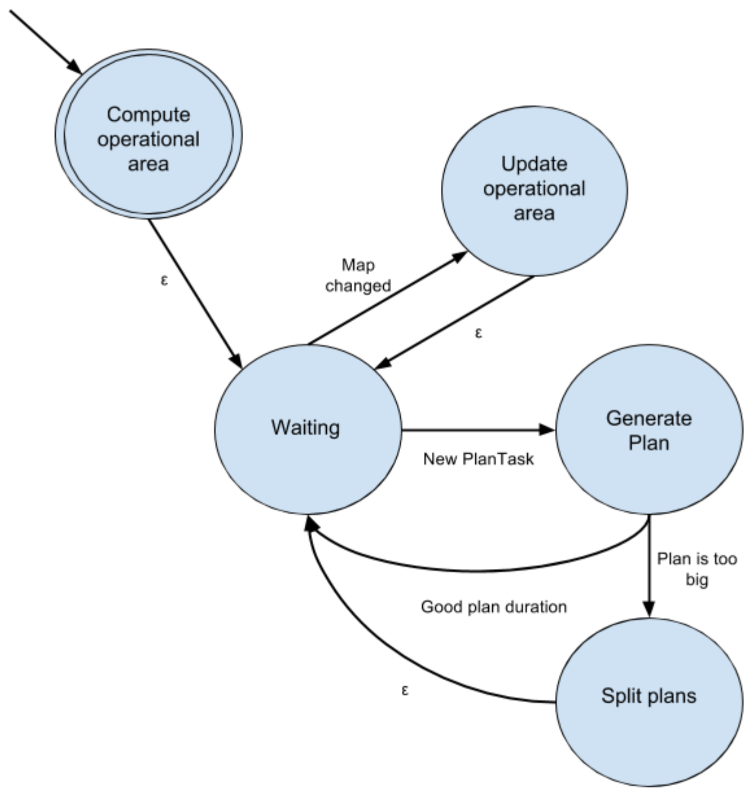


Figure 3.12: Plan generation state machine

When a new *PlanTask* type is defined, *PlanGenerator* needs to know how to handle it, so a new method needs to be created, that implements the plan generation for tasks of that type.

This module also computes and holds the *operational area* used when closing a task. The *operational area* is used to compute the safe paths necessary for the completion of this request.

3.5.1 Coverage Path Planning: SpiralSTC

Even though the implementation is modular and allows other algorithms to be used instead, offline SpiralSTC algorithm was chosen and implemented to generate coverage paths. Besides permitting a simple decomposition of the map 2.5.6 and selection of the cells to traverse, a simple path can also be generated easily by computing and following a Spanning-tree.

Most other algorithms, not Grid-based, require that after the cells to traverse are selected, an actual path needs to be generated to cover said cell. On the contrary, with SpiralSTC when a robot moves inside the cell, due to its size, it gets covered. Finally, because SpiralSTC makes the robot move around a cell's center location, it handles naturally complex situations like reaching a dead-end cell, *i.e.* a cell with no neighbors besides the one the robot just moved from, requiring it to move back again, which removes complexity from the implementation. Another major advantage of this algorithm is that the way it generates paths is in line with *LSTS* vehicles's dynamics. In no situation a path is generated so that a vehicle needs to move backwards or turn around in order to reach the next location.

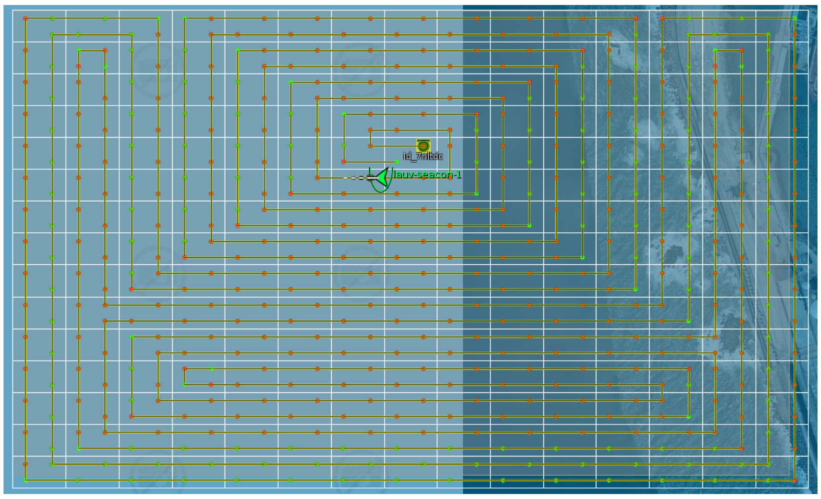


Figure 3.13: Complete coverage area plan

In this thesis we're not concerned with shaping or selecting specific areas to be covered ,hence each edge has a weight of 1 between two adjacent cells. Then, a simple *Depth-First search* algorithm [25] is employed to find the *spanning-tree.* Finally, after a plan that covers the whole area can be generated, one can split and allocate it to several robots, according to the ones available.

Figure 3.13 shows an example of a, large, coverage area plan generated using SpiraSTC's algorithm, and Figure 3.14 shows the sidescan data acquired during a coverage plan, overlayed on top of its path.

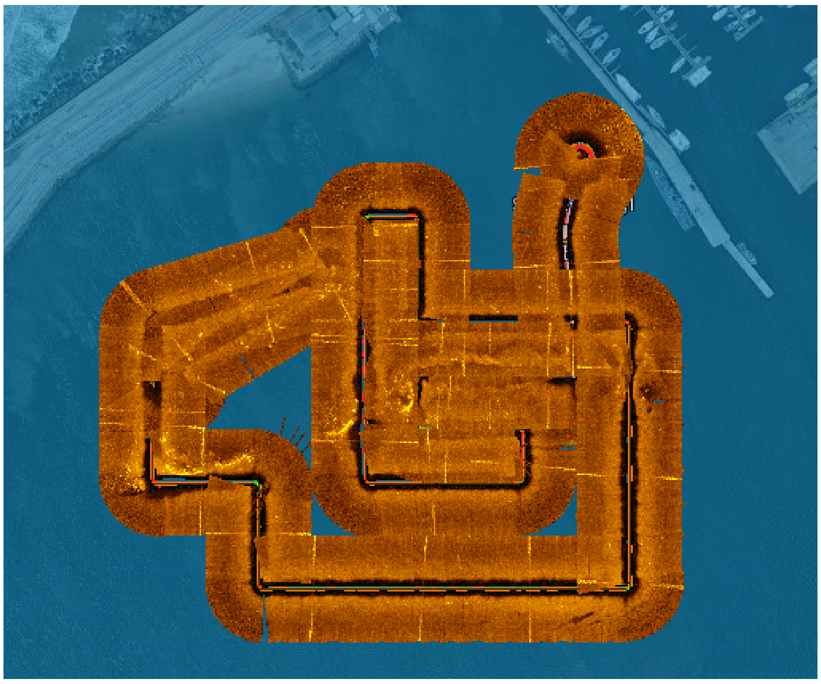


Figure 3.14: Sidescan overlay

3.6 Plan allocation

The *PlanAllocator* module is what's often called the *Planner*. Besides holding an object responsible for employing an allocation strategy, a subclass of *AbstractAllocator*, it also receives and parses PlanTask's from the *UI* module and replanning "events" from *ExternalSystemsMonitor* or other supervisor module.

When a new *PlanTask* arrives at *PlanAllocator* it uses *PlanGenerator* to generate the corresponding plan, stores it in the *PlanTask* itself, in the form of a *PlanType*, and then it's set as ready for allocation. After that, the allocator object will decide when to send the plan to a given vehicle.

It can also happen that a task needs to be allocated right away, bypassing the allocation strategy. This happens, for instance, when *ExternalSystemsMonitor* informs *PlanAllocator* that an *External system* (Ships, etc) may collide with one of the autonomous vehicles currently performing a task. In result of that, a *ToSafety* task is created (and the corresponding plan generated) in order to move the vehicle from its current location to a pre-defined safe location set by the operator.

This module is also responsible for saving and loading, any task that isn't completed. It can happen sometimes, for various reasons, that the vehicle will stop executing its

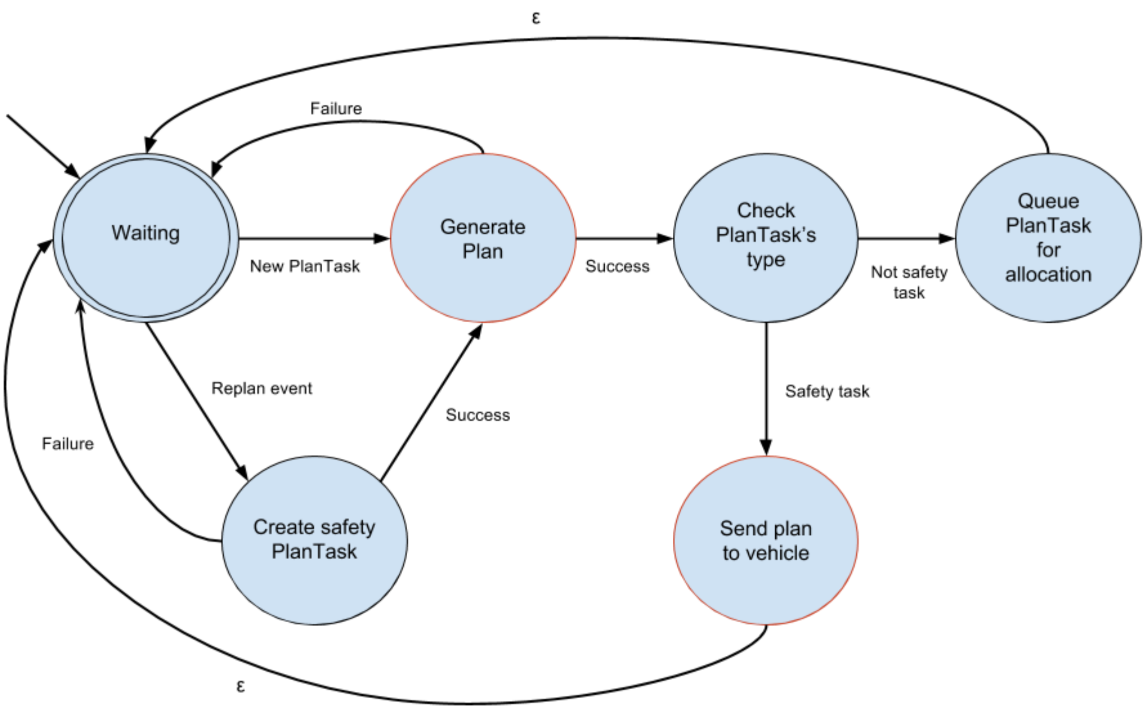


Figure 3.15: PlanAllocator state machine

current plan. If this plan originated from a *PlanTask*, then *MvPlanning* will add that task for allocation, again. In case the task doesn't get allocated or completed until *MvPlanning* closes it, *PlanAllocator* will marshal it into an XML file, so that the next time *MvPlanning* is started, the task is loaded and added for allocation. It should be noted that only essential information contained in *PlanTask* will be marshaled and any information contained in its sub-classes will be lost.

When this saved tasks are loaded, at *MvPlanning*'s start up, they will be seen as *NeptusPlan* tasks, because any information that could be used to identify it, was not saved. This is not problematic for the planner because that additional information was only used to generate the plans and, at this point, that has already happened.

3.6.1 Allocation strategy

The abstract class *AbstractAllocator* A.2 sets the common behavior for all the allocation strategies. It can be a periodic strategy, meaning that periodically it will check if there are any new tasks to be allocated and any vehicle to take them, or it can be event-based where its behavior will be triggered by events. Also, by extending this class, all sub-classes will be forced to allocate tasks to the vehicle through *TCP* connections, thus providing better feedback on the tasks' allocation status.

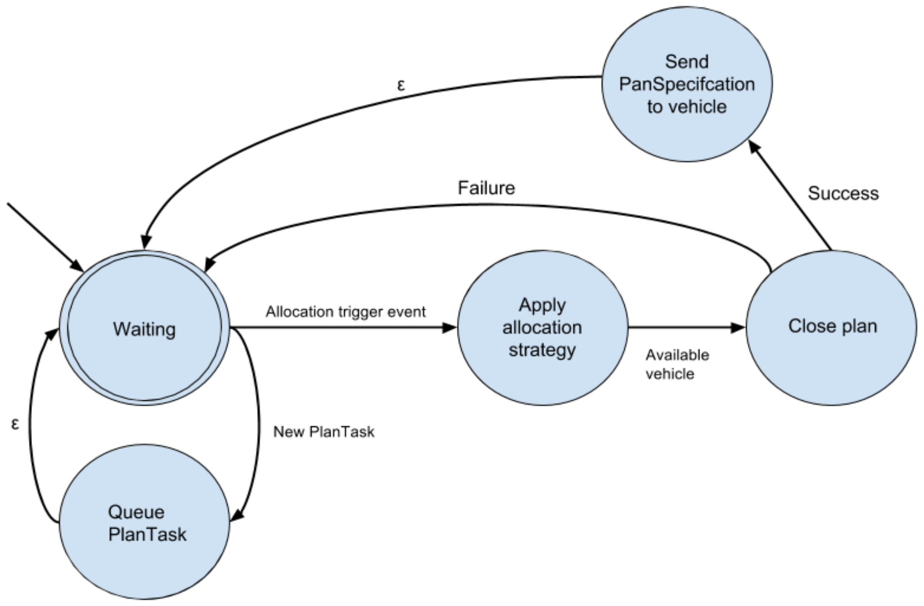


Figure 3.16: Allocation strategy's base behaviour

Before allocating a task to a vehicle, the allocator needs to check if all the task's constraints are met by the vehicle. For instance, if the task is to survey/cover an area with a sensor, the vehicle needs to have both the sensors and battery to perform the task. Then, if a vehicle is deemed as available, the allocator asks *PlanGenerator* to *close* the plan. This step computes two safe paths and adds them to the, already, generated plan: one from the current vehicle's location to the task's first location, and another one from the task's last location to the pre-defined vehicle's safe location.

Finally, the closed plan (*PlanSpecification*) is sent to the vehicle using the available means of communication, preferably *Wi-fi* and through a *TCP* connection.

3.6.2 Replanning

Currently, a replanning event consists of a simple call to a *PlanAllocator* method

public void replan(String vehicle);

that a Supervisor class makes and its behavior is the same independently of the reason why the replanning is needed. When *PlanAllocator* receives it from a module, *e.g.*

ExternalSystemsMonitor, it creates a *ToSafety* task for the given vehicle and it gets allocated to it.

One problem with this approach is that the *speed* and *Z* values are fixed. The task created makes the vehicle return to the safe location while underwater and if there's loss of underwater communications or low battery levels (among other things). We refer to it as *Replanning* because its future intent is that once the targeted vehicle can operate safely again, it resumes executing its plan, continuing from where it was left of.

3.7 Internals and utility classes

In this section the most important data structures and classes used internally, are analyzed.

3.7.1 PlanType Java Class

When a plan is created in *Neptus*, its information is stored in the form of a *PlanType* object. When a plan is sent to a vehicle, the selected *PlanType* is converted into a *PlanSpecification* message and then sent.

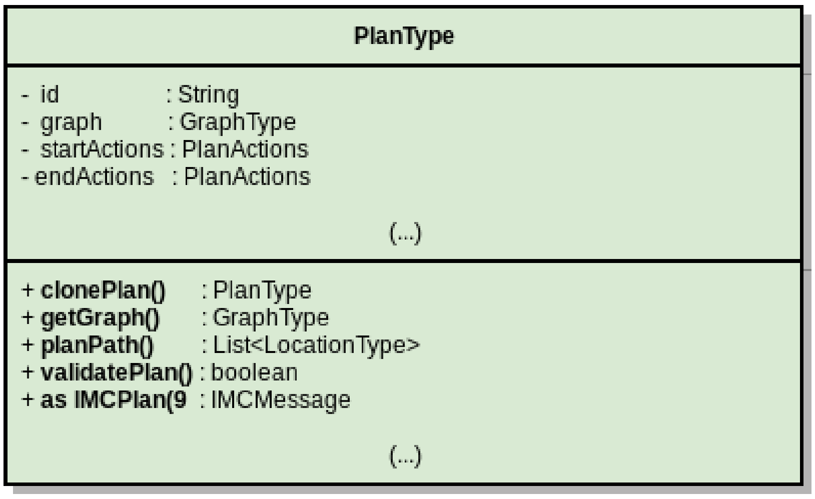

PlanType
- id : String
- graph : GraphType
- startActions : PlanActions
- endActions : PlanActions
(...)
+ clonePlan() : PlanType
+ getGraph() : GraphType
+ planPath() : List<LocationType>
+ validatePlan() : boolean
+ as IMCPlan(9 : IMCMessage
(...)

Figure 3.17: *PlanType* definition

3.7.2 Map cells

Most of the algorithms used to generate paths, from coverage area to safe paths, need a discretization of the working space. *MapCell* is an abstract class that can be used to create the atomic unit of this discretizations, the cell. By extending this class, cells of any size and shape can be created. For instance, the algorithm *SpiralSTC* uses squared cells to represent the space as a grid, while to use the *Boustrophedon decomposition*, a trapezoidal representation of the space would be needed.

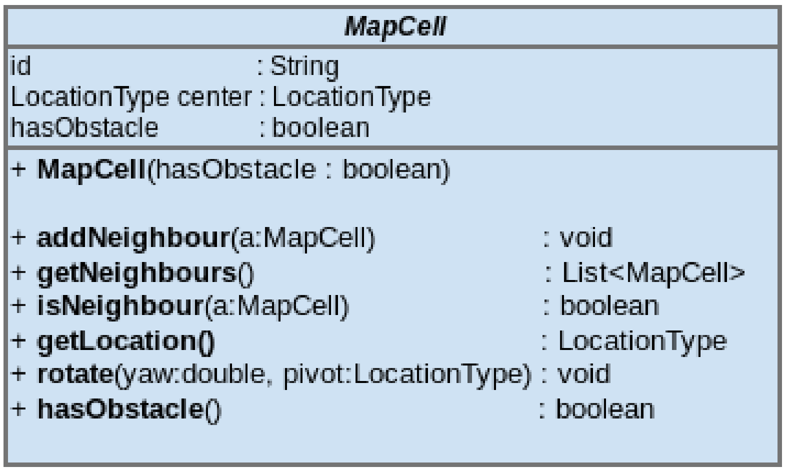

MapCell
id : String
LocationType center : LocationType
hasObstacle : boolean
+ MapCell(hasObstacle : boolean)
+ addNeighbour(a:MapCell) : void
+ getNeighbours() : List<MapCell>
+ isNeighbour(a:MapCell) : boolean
+ getLocation() : LocationType
+ rotate(yaw:double, pivot:LocationType) : void
+ hasObstacle() : boolean

Figure 3.18: UML diagram of MapCell

A *MapCell* object consists of an *id*, its center location, a list of neighbour cells and a flag that signals if the cell has an obstacle or not. When creating new types of cells, further information can be added, if needed.

3.7.3 Map decompositions

A map decomposition, *MapDecomposition* class, is simply an interface that enforces sub-classes to aglomerate *MapCell* objects, and provides functionalities to interact with them. Classes that implement it, are responsible for implementing the way the decomposition of the space takes place. *MapDecomposition* allows the other modules to use a discrete representation of the space without caring about its shape or internals.

MapDecomposition objects can be painted in the console's map, for debugging or informational purposes, so sub-classes also need to implement how such decompositions are displayed.

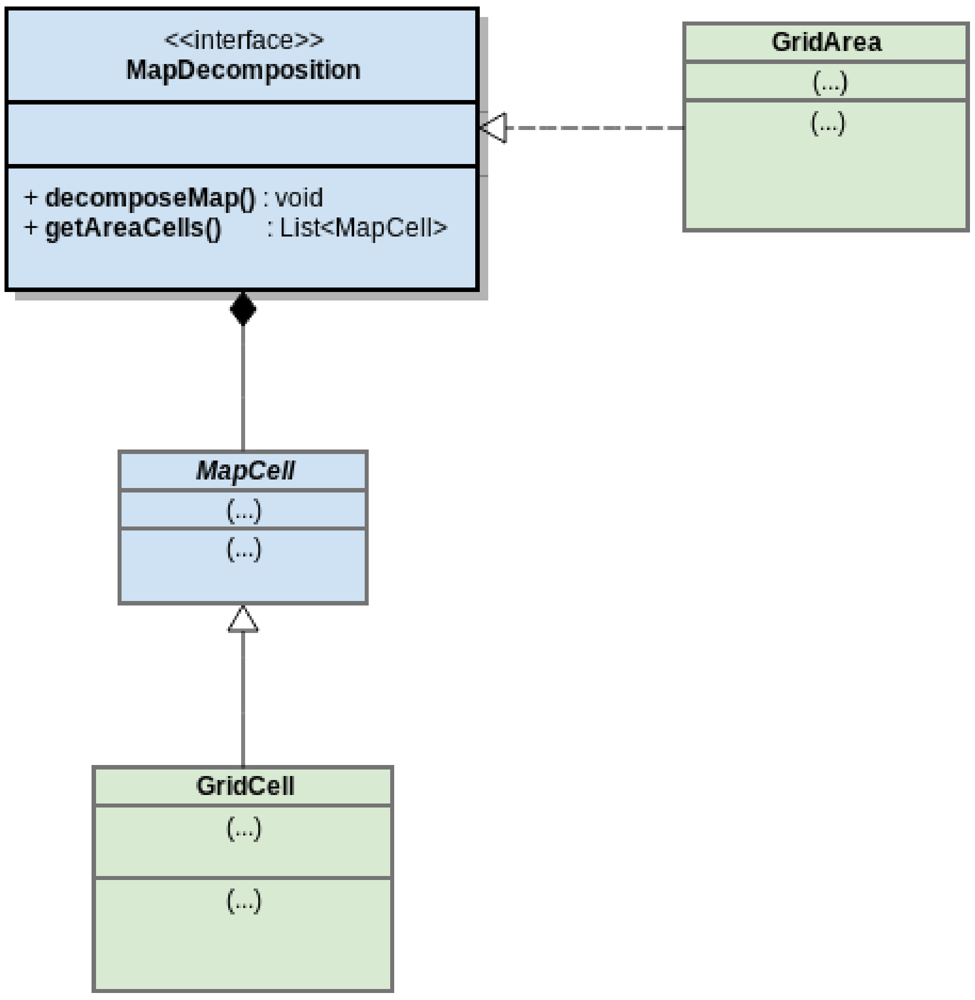


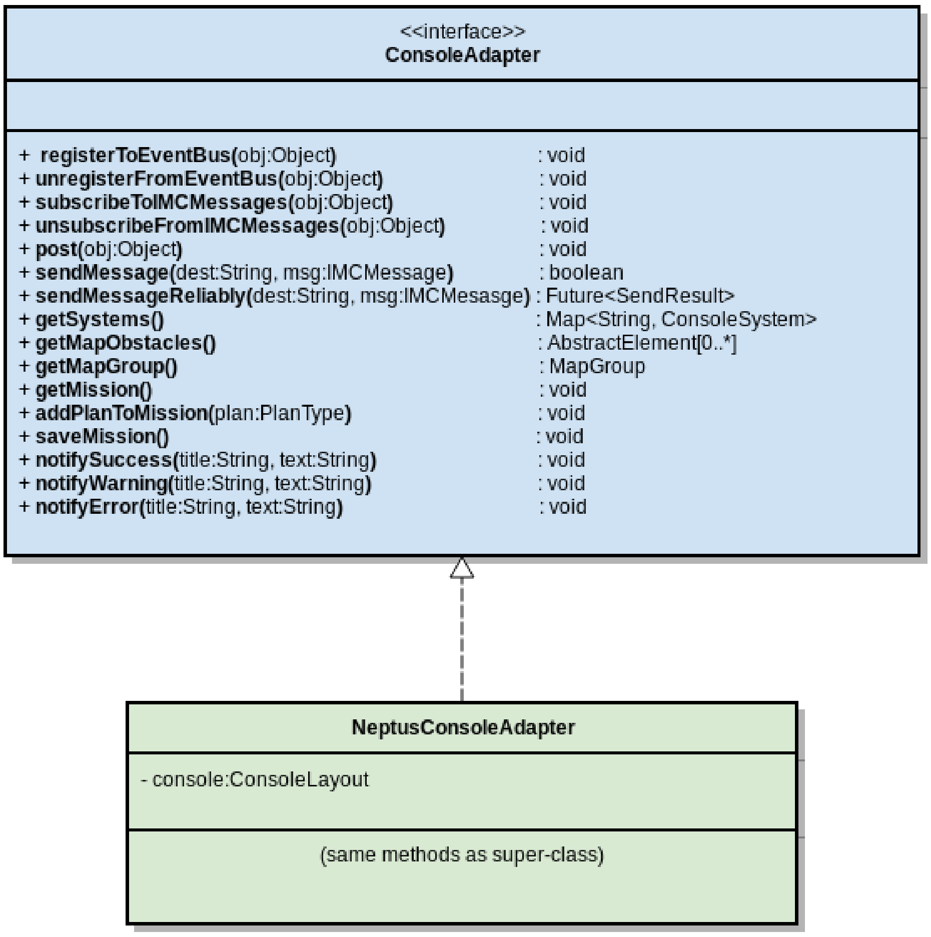
Figure 3.19: UML diagram of MapDecomposition

3.7.4 GridArea

The main map decomposition used in *MvPlanning* to represent the *operational area*, is the *GridArea* A.3 which is a grid-like decomposition, and uses *GridCell* objects, a sub-class of *MapCell*, to represent the squared cells. *GridArea* decomposition can also be used in area coverage algorithms (figure 2.13).

3.7.5 Consoles and adaptors

ConsoleAdapter is an interface whose role is to abstract the interaction of the modules with the console where the operator will be adding tasks, and where obstacles are identified. This can happen through a *Neptus* console, as well as through *Ripples*.



<<interface>>
ConsoleAdapter
+ registerToEventBus(obj:Object) : void
+ unregisterFromEventBus(obj:Object) : void
+ subscribeToIMCMessages(obj:Object) : void
+ unsubscribeFromIMCMessages(obj:Object) : void
+ post(obj:Object) : void
+ sendMessage(dest:String, msg:IMCMessage) : boolean
+ sendMessageReliably(dest:String, msg:IMCMesasge) : Future<SendResult>
+ getSystems() : Map<String, ConsoleSystem>
+ getMapObstacles() : AbstractElement[0..*]
+ getMapGroup() : MapGroup
+ getMission() : void
+ addPlanToMission(plan:PlanType) : void
+ saveMission() : void
+ notifySuccess(title:String, text:String) : void
+ notifyWarning(title:String, text:String) : void
+ notifyError(title:String, text:String) : void
NeptusConsoleAdapter
- console:ConsoleLayout
(same methods as super-class)

Figure 3.20: UML diagram of ConsoleAdapter

Currently, only *NeptusConsoleAdapter* is implemented which abstracts the interaction with *Neptus*'s consoles, *ConsoleLayout*. The set of functionalities provided by this interface range from sending messages to vehicles, subscribing to *IMC* and event buses, as well as publishing, among others.

3.7.6 External systems simulator

The supervisor *ExternalSystemsMonitor* relies on the *AIS* system to detect maritime vehicles, operated by humans, that might be crossing the working area of the mission. Because not all of this vehicles have such a system, or might not have it turned on, they will go by undetected by *MvPlanning*. Module *ExternalSystemsSimulator* allows the operator to add, by hand, *external systems* to the map, or move them, simulating any real vehicle that might be present, and this way triggering *ExternalSystemsMonitor* to check the vehicles safety. Currently, this supervisor doesn't support the definition of a *draught* and *bearing* values.

An *external system* is added to the map, as all *MvPlanning*'s interactions through the map, using *Neptus*'s plugin *MapEditor*. By adding a mark element with its id prefixed by ***ext_***, *ExternalSystemsSimulator* will identify it as a simulation of an *external system* and handle it. Likewise the *Environment* monitor, this module listens to *MapChangeEvent* events to be informed when a map object is changed.

To simulate and add an external system 3.9 to the map, one adds a *Neptus*'s mark element (first image) and give it an adequate name with the needed prefix.

After this, the *external system* can be re-positioned by moving the mark element (that is sitting bellow the system's symbol).

3.7.7 Exceptions

The following exceptions were created to handle possible errors:

- **BadPlanTaskException**: This exception is thrown when something is wrong with a *PlanTask* object that was created/instantiated, for instance, when the object is null, the task type is not recognized, or any of the data needed by *PlanGenerator* is missing or corrupted.

- **NoSafePathFoundException**: As with the previous exception, this one is handled by *PlanGenerator*. It is thrown by the implementation of the *A** algorithm in case the *operational area* is **null** or a safe path between the two given points cannot be found. This can happen both because a *MapCell* node is null or has no neighbours.

3.8 Use case

In the beginning, when doing research for this thesis, I participated in a *LSTS* mission whose objective was to deploy 2 *Noptilus* vehicles at *APDL*'s port, in order to survey its bay. Two operators would handle the vehicles, one each, and to best use the resources available, and because the area's size permitted, the area would be split in 2, one for each vehicle. The area was roughly split "by eye" and marked in the map, in order to guide the operators. Then, each operator would create a plan by using the maneuvers provided by *Neptus* and setup its parameters. While planning the mission it was clear that it was a strenuous endeavour trying to synchronize all the details between operators, guaranteeing for instance that the vehicles wouldn't collide with each other, the shore or any obstacles, and minimizing overlap between plans.

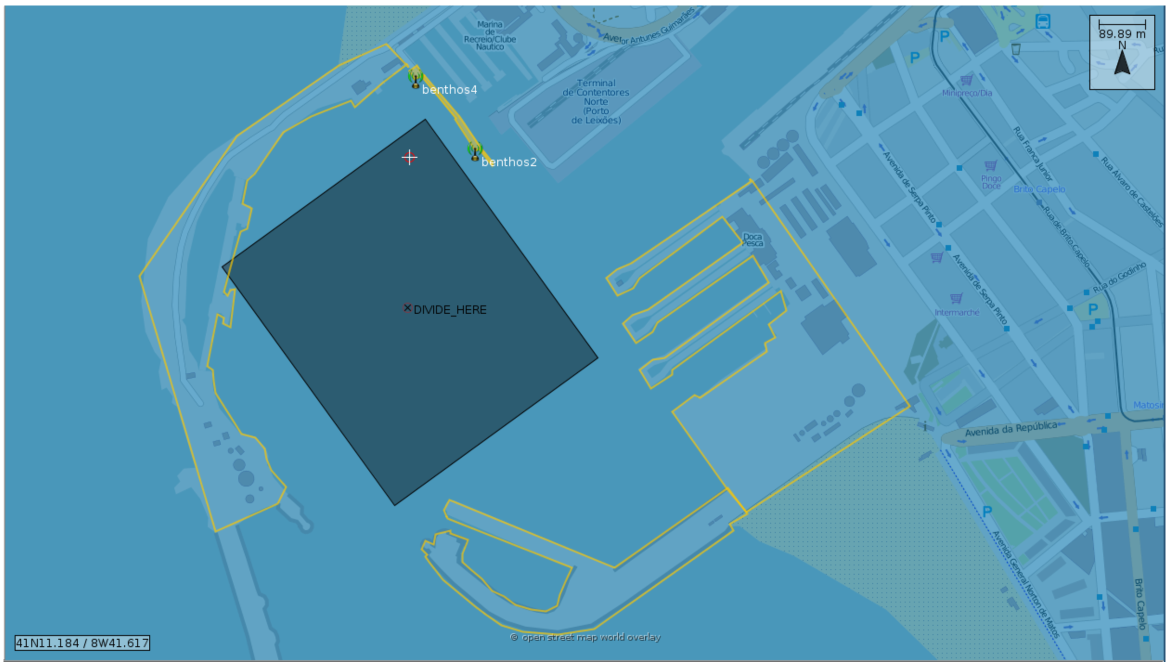

benthos4
benthos2
DIVIDE_HERE
89.89 m
N
41N11.184 / 8W41.617
© open street map world overlay
Terminal de Contentores Norte (Porto de Leixões)
Marina de Recreio Clube Náutico
Avenida da República

Figure 3.21: *LSTS* mission's area to survey

The maneuvers need to be configured according to the *sidescan* sensor used. In this case the sensor was configured to a *range* of 30 meters and a *frequency* of 770Hz. Besides this, the rows have to be spaced right, so that it exists some overlap in the sensors' readings (because of the loss of quality discussed in the Background chapter), but not so much that the vehicles take too long to perform the task while gaining nothing from data redundancy. Another problem associated with *surveys* is the accumulation of navigational error while underwater. This can be reduced by using an tactical-grade *IMU* - *Inertial Measurement Unit* which improves navigation, but still accumulates error that is typically noticeable (on *LSTS*'s missions) in areas larger than 200x300 *meters*.

In the current scenario, the area has a dimension of 485x570 *meters* and, by the way it is divided, gives each operator an area of 485x243 *meters* to survey. Because the objectives require that the vehicles move underwater, in order not to take risks of incurring too much navigation error, the operators have two options: either use several *row maneuvers* or use a combination of *goto* and *popup* maneuvers, like it is shown in figure 3.22.

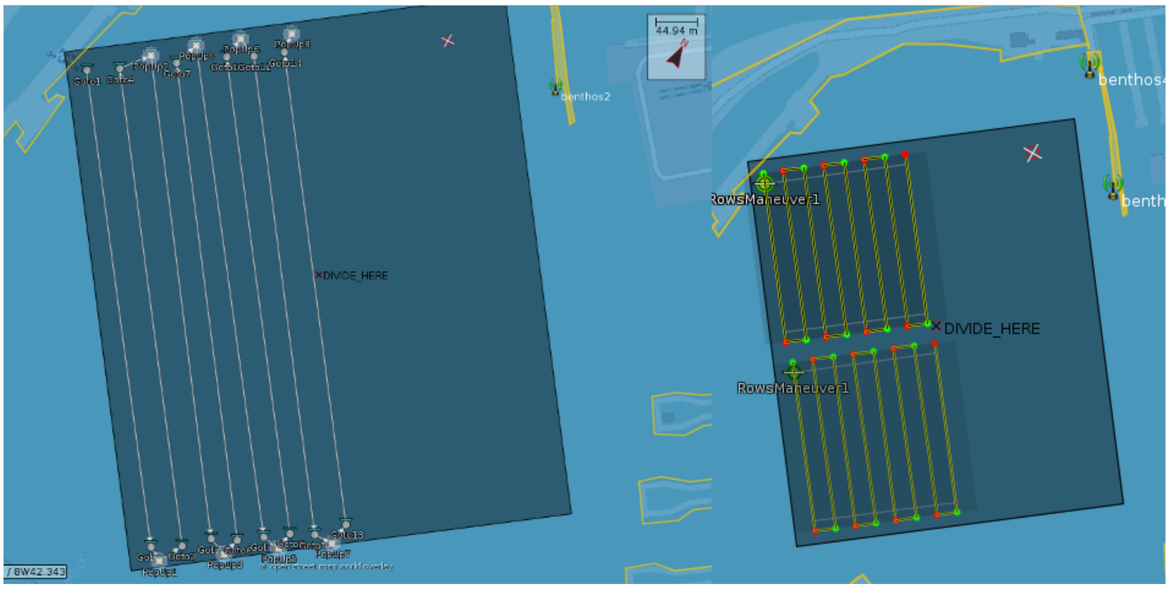
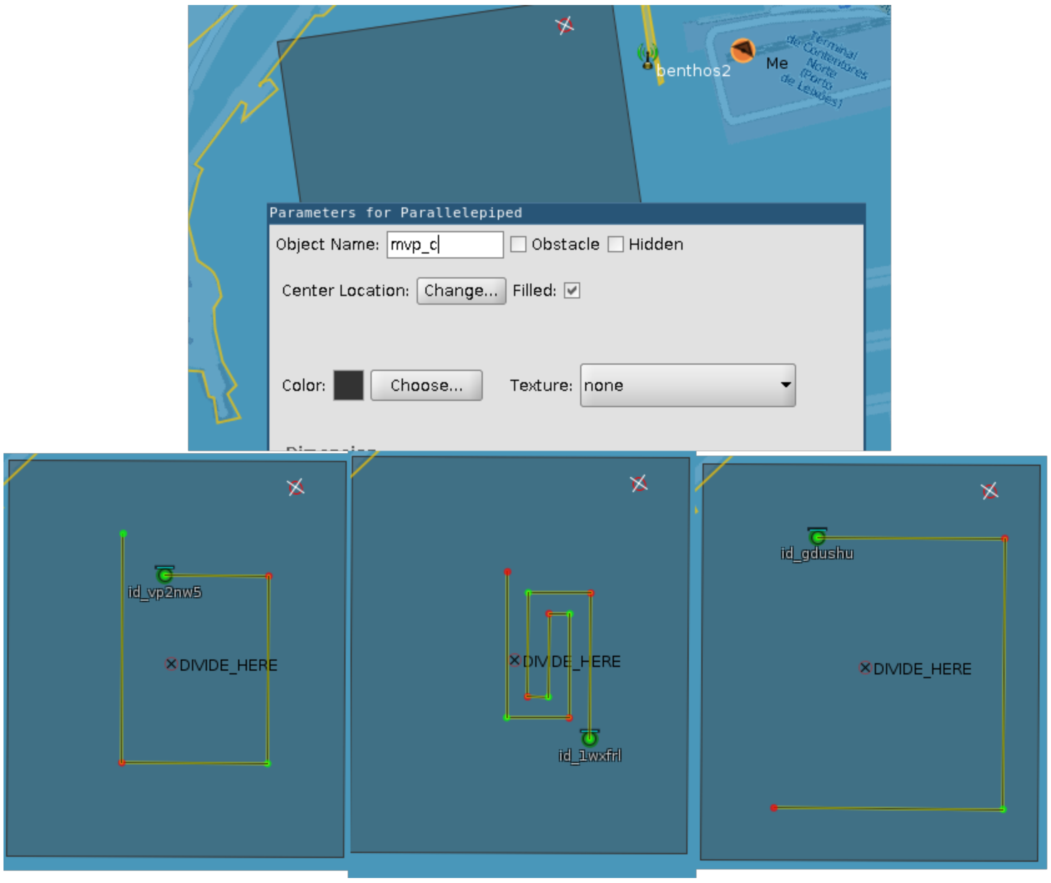


Figure 3.22: Different ways of approaching survey planning, by hand. On the left the mix of "goto" and "popup" maneuvers. On the right two "rows" maneuvers

Either way, these plans are generally created *apriori* of the arrival at the mission area, which makes this task less stressful and less error-prone. The complications occur when something in the scenario changes. Sometimes the vehicles change, and thus some of the sensor parameters might change too, some more obstacles that need to be accounted for, or simply the area's positioning or dimensions are not correct. All of these problems have happened and are prone to happen again while there's a Human operator as the main piece on this mission's stage, due to its complexity.

MvPlanning and its algorithms for coverage area plans reduce all these steps to simply adding the obstacles and target area to the map (with an appropriate ID). With this, the operator doesn't have to care, also, about navigation error because larger plans are split into smaller ones (typically 15 minutes long) reducing the chances of error accumulation. Furthermore, *MvPlanning* allows the operator to change the



Parameters for Parallelepiped

Object Name: mvp_c ☐ Obstacle ☐ Hidden

Center Location: Change... Filled: ✓

Color: ■ Choose... Texture: none

Figure 3.23: Some of the plans generated by *MvPlanning*

payload/sensors and parameters of the plan very quickly, whereas with the typical planning process the operator needs to apply the changes to all waypoints where this changes were needed. *MvPlanning* reduces many of the points of failure, due to human weaknesses, that occur during planning.

Chapter 4

Experiments

4.1 REP16 - Rapid Environment Picture

REP is an annual exercise organized by LSTS and the Portuguese Navy whose purpose is to test and demonstrate the cooperation and synergy between aerial, surface and underwater robotic vehicles, with a high focus on multi-vehicle operations both homogeneous and heterogeneous in nature. Some exercises and experiments developed at REP16, to name a few, were:

- Search & Rescue with an UAV equipped with IR camera;
- Control and Monitoring through multi-hop acoustic network;
- Swarm planning on AUVs;
- AVS + UAV as communication relays for AUVs.

At REP16 the exercise named "Multi-Vehicle's Network planning and control" had the following objectives:

1. Access how well *MvPlanning* aided an inexperienced operator to create objectives and have multiple vehicles performing them;
2. Validate *MvPlanning*'s architecture in a real-world scenario;

3. Access SpiralSTC algorithms' performance on scenarios with a high number of obstacles, like navy base pontoons and its stationary ships. Also access the vehicles performance while traversing the spiral paths created;
4. Understand, in a real-world scenario, the difficulties of replanning, with *MvPlanning*, in case a mission's requirements changes;
5. Fix any issues found while using *MvPlanning* in the field.

The experiments were divided into 2 days, the 11th and 18th of June, and the areas of operation were at Alfeite's Portuguese Navy Base and off Sesimbra, respectively. To note that even though in the survey plans the vehicles had their sidescan sensors collecting data, its validity was not the purpose of any of the exercises.

4.1.1 Experiment 1 - Alfeite's navy base

The following vehicles were involved in the experiment:

- LAUV Noptilus-1
- LAUV Noptilus-2

Five tasks were added to the map, *a priori*, so that *MvPlanning* could distribute them: 4 points to visit (with no sensor) and 1 area of 218x224 meters to cover with a sidescan sonar. The scenario is displayed in Figure 4.1

All the map objects colored yellow are obstacles or restricted areas signaled by the operator; the marks prefixed by $mvp\backslash_{visit}$ are the locations to visit and the black squared area is the area to survey. The green triangle signals a docked ship (whose area was exaggerated for safety reasons), which was only detected after the arrival at the operation area, hence added *a posteriori* to the mission.

For the survey mission a profile was created with the following parameters:

- **Speed**: 1.0 m/s
- **Z**: 0 meters of depth
- **Sidescan sensor**:

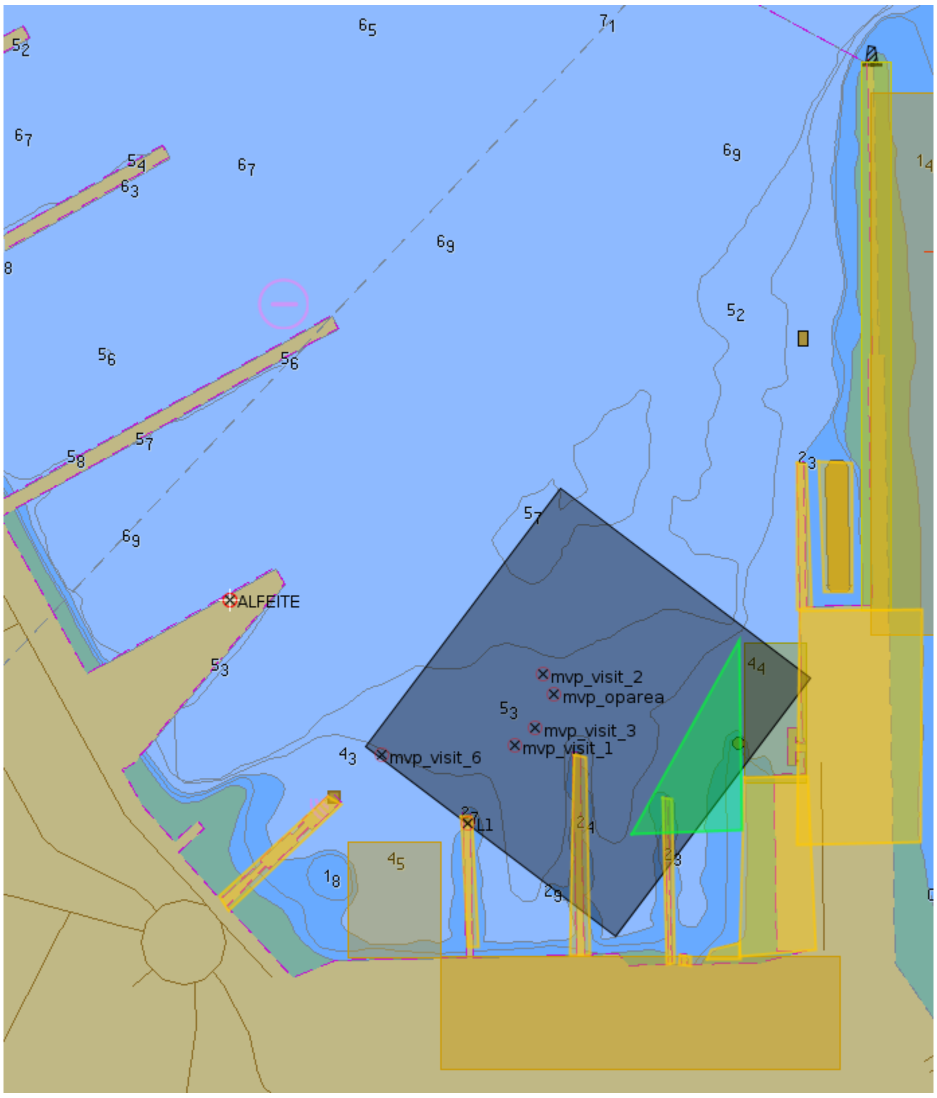


Figure 4.1: Scenario of experiment 1 at Alfeite's Port

- **Range**: 30 meters
- **Frequency**: 770 Hz

and the tasks to visit the other locations had the same parameters except for the sidescan sensor.

After the vehicles were configured and ready to start the experiment, *MvPlanning* received all tasks and generated the respective plans. At this stage it was clear that SpiralSTC, as expected, was not a very good choice to generate a coverage plan in such a scenario with lots of obstacles:

From the area to survey, only a small fraction was considered "usable" by the algorithm

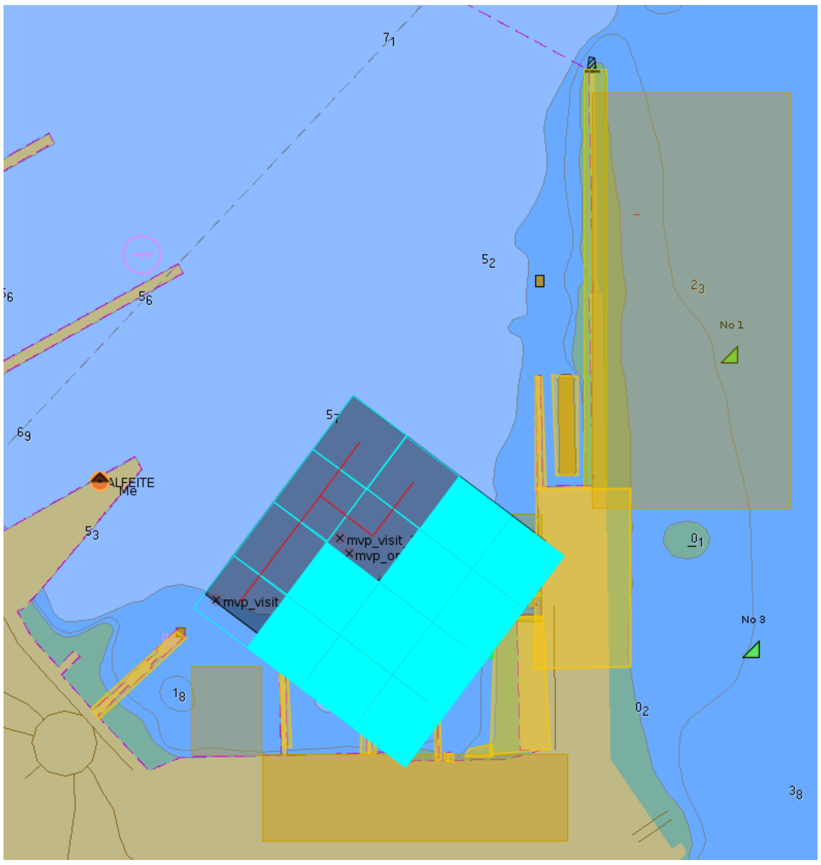


Figure 4.2: SpiralSTC: Wasted space, in blue.

and to be covered by the vehicles. The solution here was to reduce the size of the cells from 60 to 15 meters, meaning that more data redundancy would be achieved, but the vehicle would have to perform tighter curves (which could be achievable or not, depending on the vehicles' dynamics). Smaller cells also reflect in greater number of generated plans for the same area, and more time that the vehicles are deployed with their sensors draining battery.

Throughout the experiment *Noptilus-2* kept entering in *error mode* and right after, in *service mode*, due to internal errors (not due to *MvPlanning*). As a consequence *Noptilus-2* kept stopping and consuming new tasks that would get allocated to it. This happended because at this point *MvPlanning* was not yet able to understand when a plan was completed successfully or not. In the end *Noptilus-1* was able to perform only one task while the other vehicle started and stopped all the other ones, helping, at least, validate some of *MvPlanning*'s features and modules such as *VehicleAwareness* and *PlanAllocator*. During this mishaps *VehicleAwareness* identified correctly the states of *Noptilus-2* (as available or unavailable) and *PlanAllocator* selected new tasks to allocate to it, as it should.

Finally, even though changing SpiralSTC's parameters helped improve the amount of area that could be covered, the tighter curves and paths proved to be difficult for *Noptilus-1* to execute which would most likely reflect on the quality of the sonar data collected.

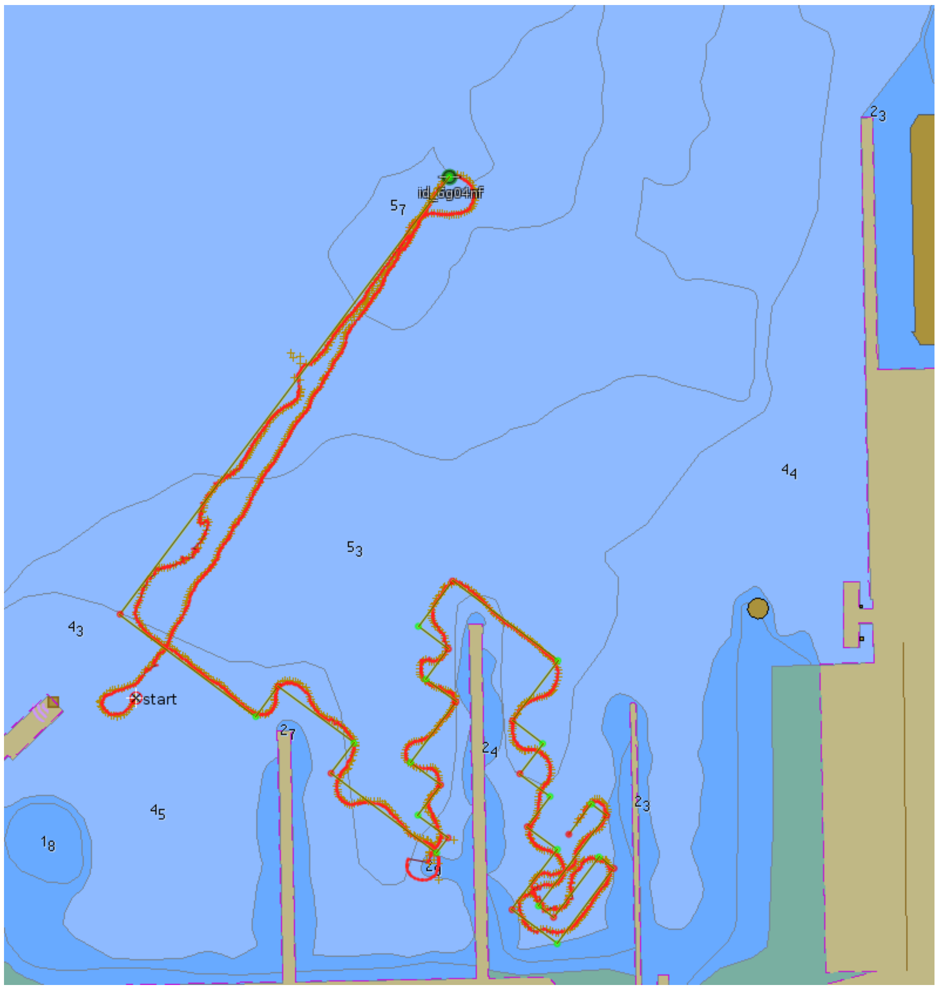


Figure 4.3: One of the plans generated at Alfeite, as executed by *Noptilus-1*

4.1.2 Experiment 2 - Sesimbra's coast

The second experiment conducted at Sesimbra's coast, intended to conclude the previous experiment by having both vehicles survey most of the area selected, test bug fixes such as task-interruption handling, and test safe paths (which were implemented but not being deployed until then). The vehicles deployed were:

- LAUV Noptilus-1
- LAUV Noptilus-3

There were no obstacles in the area of operation (besides the coast), but there was a number of vessels crossing it which encumbered the operator with being attentive to the vehicle's safety. When an incoming ship was estimated to cross the UAVs path, the operator had to pause *MvPlanning*, stop the vehicle's plan and send it back near the coast. After the area was clear *MvPlanning* would be resumed and the interrupted tasks handled, by setting them for allocation again.

This experiment helped understand that, with *MvPlanning* the operator needs to perform an additional step when handling ships crossing the vehicles' path, which is to pause *MvPlanning*, instead of just taking control of the vehicle. The modules *ExternalSystemsMonitor* and *ExternalSystemsSimulator* were then created in order to test new ideas and ways of dealing with this scenario. The profile used in the survey task was the same as in the previous experiment, except that now the vehicles would be performing the plan at 3 meters of depth, instead of at the surface.

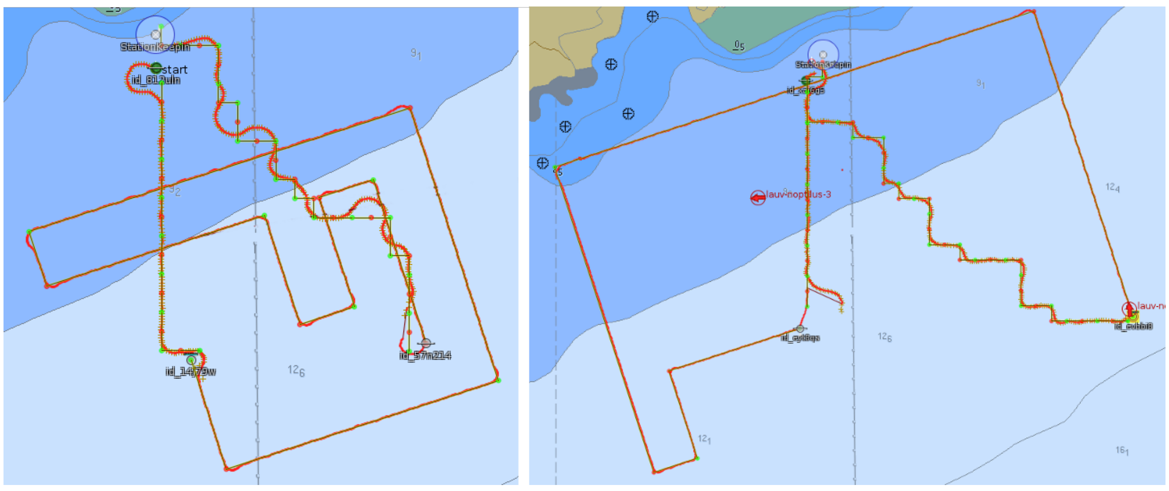


Figure 4.4: Part of the paths covered by *Noptilus-1* and *Noptilus-3* at Sesimbra's coast, already with safe paths.

When starting a plan the vehicles moved through a safe path to the first location in the plan, generated by *MvPlanning*, and at the end of it another safe path was traversed, but this time to the safe location set by the operator.

4.2 Discussion

After many experiments (real-world experimentation and simulations) some improvements to *MvPlanning* were identified. The results and possible improvements discussed in these sections entail more direct and small changes to the architecture, whereas the ones at Future work chapter do need more research and might need more developments.

4.2.1 Tasks

4.2.1.1 Awareness interfaces

Some more information could be displayed to the operator, such as what's the state of the currently connected vehicles, which tasks are available and their state (*e.g.* allocated, not allocated, in error or finished). Even though this information is important for the operator's awareness, special care needs to be given on how it would be displayed. A recurrent problem of software like *Neptus* is cluttering of the console with information and graphical windows, to a point that it gets in the way and the operator gets overwhelmed.

4.2.1.2 ToSafety task's depth

Like it was discussed in Chapter 3, when a task's plan is generated, safe paths are computed guaranteeing that the vehicle moves without colliding with known obstacles. Currently, it is not possible to define at which depth the vehicle should travel during this plans, it being a *hard-coded* value of depth 0.

There should be a way of deciding which value to set, either autonomously by *MvPlanning*, or by the operator, for instance, in a profile's file.

4.2.2 Error handling

There are a lot of variables that the operator and *MvPlanning* can't control. Sometimes the vehicle goes into an error state because of its environment, *e.g.* water currents too strong, unexpected obstacles, *etc.*, or internal mishaps, *e.g.* some *DUNE* task/module or sensor that failed, wrong configurations, *etc.*. For safety reasons, the vehicle cannot be left unattended after it stops the assigned task. It might not be problematic if the vehicle is within radio range and there's already a new task to be allocated to it, but that cannot be guaranteed.

4.2.2.1 Interrupted tasks

MvPlanning's way of dealing with these errors is to send the vehicle back to its assigned *safe location*, by means of a *ToSafety* task, and then proceed with task allocation, for that vehicle, if any tasks are available. This is a simple and safe strategy, that works, but not very optimal. While the vehicle is traveling to the *safe location* it won't be able to receive any other tasks because its operational state is *Maneuvering*, hence *MvPlanning* will see it as unavailable. The first improvement to be made should be:

- Differentiate when the vehicle is doing a *ToSafety* task, from other tasks. This way, while the vehicle is en-route, back to base, *PlanAllocator* can try to allocate other tasks.

There's, still, a problem to this solution: if the vehicle stopped due to errors it can happen that it will stop again, every time a new task is allocated to it. The proposed solution is:

- Keep a counter on how many times a vehicle has failed. In case it failed more than a pre-defined value, ground the vehicle and consider it *unavailable*. This can be done permanently or until the operator fixes the issues (if they are fixable). *MvPlanning* could consider that the problems are fixed when the vehicle is restarted (hard or soft reboot).

Another interesting feature to implement would be the ability to resume a plan where it was interrupted. For instance, if a *survey* was interrupted with 63% of completion it would be more efficient to task a vehicle to survey the other 37% of the area, instead of all of it.

In case the vehicle's survey was on the surface, *PlanAllocator* could use the information of tasks' completion, by listening to *PlanControlState IMC* messages that are sent by vehicles, and adapt the *PlanTask* before sending it for allocation again. *Sidescan* and *Multibeam* surveys are not usually performed at the surface, meaning that *PlanControlState* messages would not be received by *Neptus*, defeating the previous solution. In this case the task completion would need to be estimated based on the known conditions, *e.g* of speed, currents and elapsed time.

4.2.2.2 Partitioned areas to cover

In case the operator defines an area to cover that has an obstacle partitioning it, *MvPlanning* will only consider the first partition and ignore the other(s). This is inefficient and forces the operator to define n separate areas in order to accomplish the coverage of the whole area.

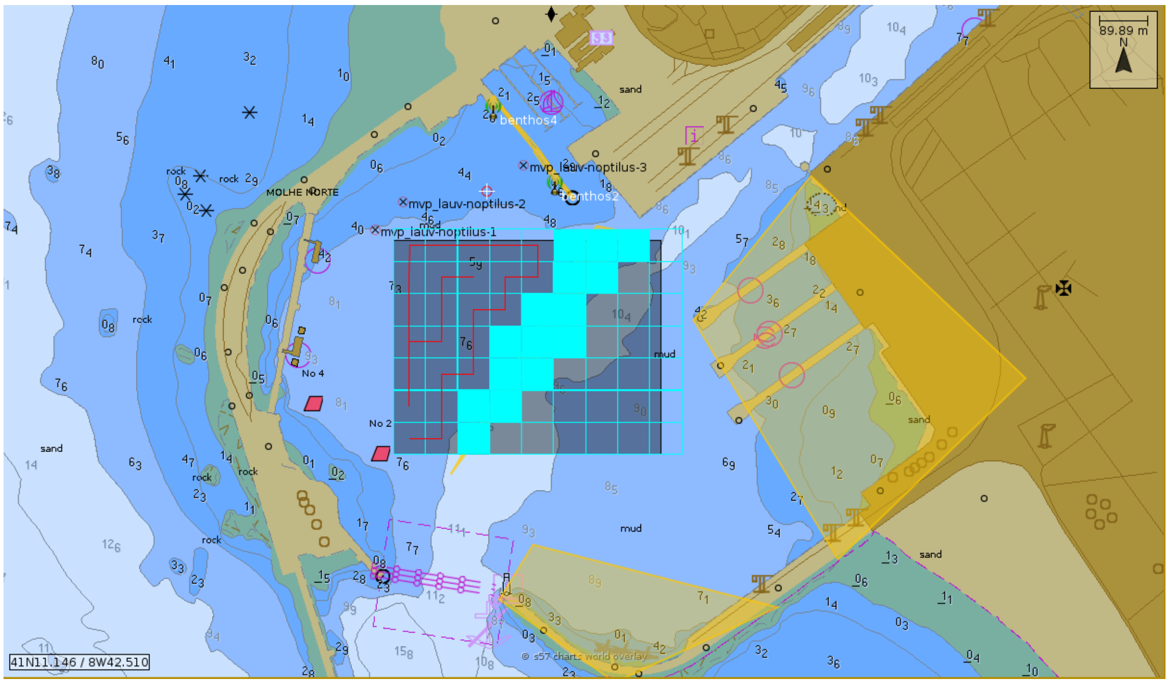


Figure 4.5: Area to cover, split in two

For this reason, *MvPlanning* should understand when an area is partitioned and then create several tasks to cover them. The image above shows, in blue, an obstacle partitioning an area to be covered, and at red the *spanning tree* used to generate *coverage area* plans, only on the first partition.

4.2.2.3 Tasks outside the operational area

If the operator defines a task that in some way makes the vehicle move outside of the operational area, *MvPlanning* will stop plan generation but no warning or notification will be displayed, except for an exception that is thrown. During simulation and testing this proved to be a bad solution.

Chapter 5

Conclusions and Future work

5.1 Conclusions

This dissertation addressed the problem of planning and coordination of multiple autonomous vehicles. The objectives defined at 1.3 were achieved and the solution was tested both in simulated and real-world scenarios 4.2.1.1 helping to validate the implemented architecture and understanding which changes and new features needed. In order to accomplish the objectives the following milestones were achieved:

1. Study of LSTS's toolchain;
2. Design and validation of *MvPlanning*'s architecture;
3. Awareness about capabilities from the available vehicles (creation of profiles 3.1.2 and VehicleAwareness);
4. Implementation of a Round-robin allocator to allocate tasks to the available vehicles 3.6;
5. Development of a simple UI for the operator to create high-level objectives;
6. Create facilities to allow more variety of high-level objectives, such as Coverage area tasks 3.1, and and their generation;
7. First round of real-world scenario's tests, to validate work developed until then;
8. Creation of task's restrictions and safeguards such as safe-paths and task's constraints, and additional monitors and supervisors 3.4;

9. Final round of real-world tests at REP16.

Finally, as it happens with most scientific and engineering work, there's always improvements to make, features to implement and new ideas to research and test. The work presented in this dissertation is no exception. The following sections provide some discussion on future work.

5.2 How much automation is too much?

Further testing needs to be made in order to access if the automation implemented defeats in some way the purpose of *MvPlanning* of improving the operator's capability in handling multiple vehicles during a mission. After thorough use of *MvPlanning* by different operators in different scenarios, it's expected that some of the flaws or missing features are detected and it can converge to the ideal solution.

Such experiments, initially, would be structured very much like the ones for this dissertation: 2 to 3 similar vehicles performing several tasks, in order to get all the operators accustomed to *MvPlanning*'s workflow. Further tests could, then, jump to more complex scenarios such as usage of multiple **heterogeneous** vehicles.

5.3 Operator interface

The biggest improvement to be made concerns the interaction between the operator and *MvPlanning.*

5.3.1 *MvPlanning* map editor

Currently the operator needs to interact with *MvPlanning* by means of *Neptus*'s *MapEditor* and having to use specific feature *id*'s prefixes, such as *mvp_* and *ext_*, to do it. This calls for a specialized map editor to facilitate the adding of map objects related to *MvPlanning.* The prefixes would still be used to differentiate *MvPlanning*'s map objects from others (since they use the same *API*) but only internally. Some of the objects available would be:

- Cover area: For coverage area tasks;

- Visit point: Create task to survey a single location;
- External system: Signal an external system's position, heading, *etc.*;
- Area obstacle: Obstacles whose area is significant;
- Path obstacle: This would allow the operator to "draw" an obstacle freely by means of line segments.
- Mark obstacle: Obstacles that can be represented as a point, *e.g* buoys, fishing traps, stationary ships.

5.4 Tasks

5.4.1 Actions

Even though most of the tasks share the same behaviour: start with a safe path to the initial point (*FollowPath* maneuver), execute the task-specific maneuver, finish with a safe path to the *safe location*, it might make sense to have more complex tasks. Perhaps a vehicle needs to upload data in a location *l2* after surveying location *l1.* The current way of achieving this would be to create two tasks: one to make the vehicle wait at *l2* and another one to survey *l1.* The problem with the current approach, though, is that because the goal is divided into two tasks, and nothing guarantees that the tasks will be allocated in the correct order to the vehicle, or at all. Another problem is that if more than one vehicle is able to perform the tasks, then it could happen that each task is allocated to separate vehicles, instead of just one as desired. *Temporal planning* (5.6.1) could solve the *out-of-order* allocation but not the problem of each task being allocated to different vehicles.

A possible solution would be to describe what each task entails, *i.e.* the sequence of maneuvers that the task consists of. This description would then be mapped to a list of actions, held by each *PlanTask* type, and "parsed" by *PlanGenerator* in order to generate the plan according to the description. This description could be written in an *XML* or a formal language like NVL [26]. For instance, the *CoverageArea* task could be described by:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Task Id="CoverageArea">
```

```
    <Action id="SafePath"/>
    <Action id="CoverageArea"/>
    <Action id="SafePath"/>
</Task>
```

while the more complex task's description could be:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Task id="CoverageArea">
    <Action id="SafePath"/>
    <Action id="CoverageArea"/>
    <Action id="SafePath"/>
    <Action id="Wait" duration="5"/>
</Task>
```

Such a solution also eases the process of creating a new *PlanTask*'s. In this iteration of *MvPlanning* if a new *PlanTask* is added, its behavior has to be programmed and hard-coded, whereas when defining a list of actions, at most, what needs to be programmed is the specific maneuver of each task type.

5.4.2 Allocation intermediate task

There might be some cases where a vehicle finishes its plan in a location where it doesn't have communication range to the base, and possibly with *MvPlanning*. As this location is known (because the plan is known) an available vehicle, for instance an *X8* aerial vehicle (Section 2.2.1.2), could be tasked with going to that location and allocating a task to another vehicle. This way there would be no need to wait for the vehicle to be in communications range to allocate a new task.

5.4.3 Loiter task

A *PlanTask* where the vehicle is tasked to *loiter* (an actual *Neptus* maneuver) in a given location for a certain amount of time, would be a requirement for plans such as networks' range extension or even *surveys* . The latter scenario could be, for instance, to film or photograph small shipwrecks whose location is known.

5.5 Improved environment awareness

Currently all the obstacles need to be added by the operator. This has proven, during missions and tests, to be cumbersome, error-prone and not an optimal solution, *i.e.* some space is wasted.

Using *Open Street Maps* XML API, the obstacle detection could be improved, by passing most of this work to the *API.* The operator would still need to add some obstacles by hand, for instance, buoys, garbage, stationed ships/boats, *etc.*, and would still need to validate that *MvPlanning* parsed correctly the obstacles' geometry.

Another improvement that could be added to the *Environment* module would be a parser for *S57* nautical charts. By extracting information about the depths of the area of operation, areas where the vehicle's operational Z (defined in the task profile) was invalid, could be filtered and the plans adjusted in accordance. *E.g.* If the vehicle's profile defined a Z value of 3 *meters* of depth and certain locations of an area to cover had a maximum depth of 2 meters, they would be set as obstacles. This feature could be coupled with *Neptus*'s tides level estimator.

Also, a way of defining obstacles' *depth* or *altitude* would prove to be useful. Some obstacles are underwater while others are on the surface, and in the latter case, the obstacle would not be a problem for a vehicle operating at 3 meters of *depth.* It would lead to safer and more efficient plans, and, in some cases, better data quality (the vehicle would turn less).

5.6 Planning

5.6.1 Temporal planning

One flaw of *MvPlanning* is that *PlanTask*'s are decoupled from time, not allowing the operator, for instance, to create tasks that have a topological ordering (*e.g.* Task A cannot be executed before Task B) By not taking into account task's completion time (during the planning phase) it's also not possible to use time as a planning metric, *e.g* sort plans so that the shorter ones are allocated first. The practical example given in section 5.4.1 is a clear example of tasks that are time dependent: data collection and data dissemination (to the operator). *Temporal planning* could be easily achieved by coupling *MvPlanning* with automated planners such as *LPG* or *Europa*, which *LSTS* already has experience with.

5.6.2 Future planning

The advantage of an *a priori* approach is in the amount of information available during the planning and coordination phases, either to optimize its decisions or make new ones. At *LSTS* missions the operator(s) might be forced to be onboard a boat (or at least have one in the water) because if the operational area is big enough and the objectives are spread, the vehicles might lose *Wifi* or *acoustic* communication. With *MvPlanning*, there's the safety that comes from ensuring that the vehicle won't stop after task completion, and knowing that it will travel to a position where better communication conditions exist (if the operator defined the *safe locations* correctly). This, again, may cause inefficiencies because if the majority of tasks happen further from the operator, the vehicle will be traveling great distances going back and forward, wasting battery.

As this is information that is available to *MvPlanning* during its life-cycle, it could make use of other vehicles, such as *UAVs* to perform out-of-sight task allocation to *AUVs*. Knowing that an *AUV* will be available at time x and will be out of reach of communication, a *UAV* could be tasked to fly to the location where the *AUV* is expected to be, at x, and allocate a task to it. An alternative approach would consist on tasking the *UAV* of extending the network (if possible) by flying in circles in-between the vehicle(s) and the operator. Besides the implementations at *MvPlanning*, *i.e.* in *Neptus*, it would also require changes in *DUNE*. Either:

- A *DUNE* task to run on the *UAV* (in the scenario above) to send a *PlanSpecification* message to the *AUV*;

- Change *UAV*'s networking configuration so that it could behave as a network range extender.

  Besides these features, *MvPlanning* would also need to support temporal planning, which is already discussed in this chapter, and a new *PlanTask* to assign the target vehicle to send plans to other vehicles.

5.6.3 Online Planning

MvPlanning follows a paradigm of offline planning, making it more fragile to dynamic environments. For instance, if an operating vehicle enters in error state somewhere where it has no communication with the base, it will stay adrift (in case of a maritime vehicle) or move in a straight line (in case of an aerial vehicle), because *MvPlanning* won't be able to handle the situation.

A possible solution would be to take on a hybrid paradigm and allow online planning, inside the vehicles. This way, even though it could occur problems of synchronization of information, some decision-making role would be passed on to the vehicles, and perhaps make the planning and coordination process less bound to static environments. For example, to tackle the problem described above, the vehicles could have a DUNE task on-board with information regarding the environment, *e.g.* operational area, obstacles, etc., (that could be synchronized with *MvPlanning* or not) and generate and execute a *ToSafety* task, taking the vehicle back to its defined safe-location.

5.6.4 Automated Planners

Like mentioned already *MvPlanning* could be coupled with automated planners in order to solve more complex problems. Both planners suggested next have been already tested and used by *LSTS*.

5.6.4.1 LPG

LPG is described by the authors as a fast planner that works by using local search to solve planning graphs, and is based on *Walksat*, an algorithm to efficiently solve *SAT-problems*. It can solve both problems of plan generation and plan adaptation. For *LPG* to be used in *MvPlanning* the planning problems would have to described in *PDDL* and a "wrapper" class, implemented as an allocation strategy, would have to be created.

5.6.4.2 Europa

Europa - Extensible Universal Remote Operations Architecture, is a planning framework developed at *NASA* [27], under *NASA's Open Source Agreement (NOSA). Europa* is generally used by embedding it in a host application. The planning and scheduling strategies it provides follow a Constraint-based Temporal Planning paradigm. It has been thoroughly used in observation scheduling for the *Hubble Telescope*, and autonomous controls of several spacecrafts such as *MER* - Mar Exploration Rover (Spirit and Opportunity). It's also used in the *TREX* framework, used in *AUV*'s planning, which itself is used by *LSTS* for *in-situ* planning in its vehicles. Among others, one of the most advantageous features of *Europa* is that it permits a *Mixed-Initiative* paradigm to planning.

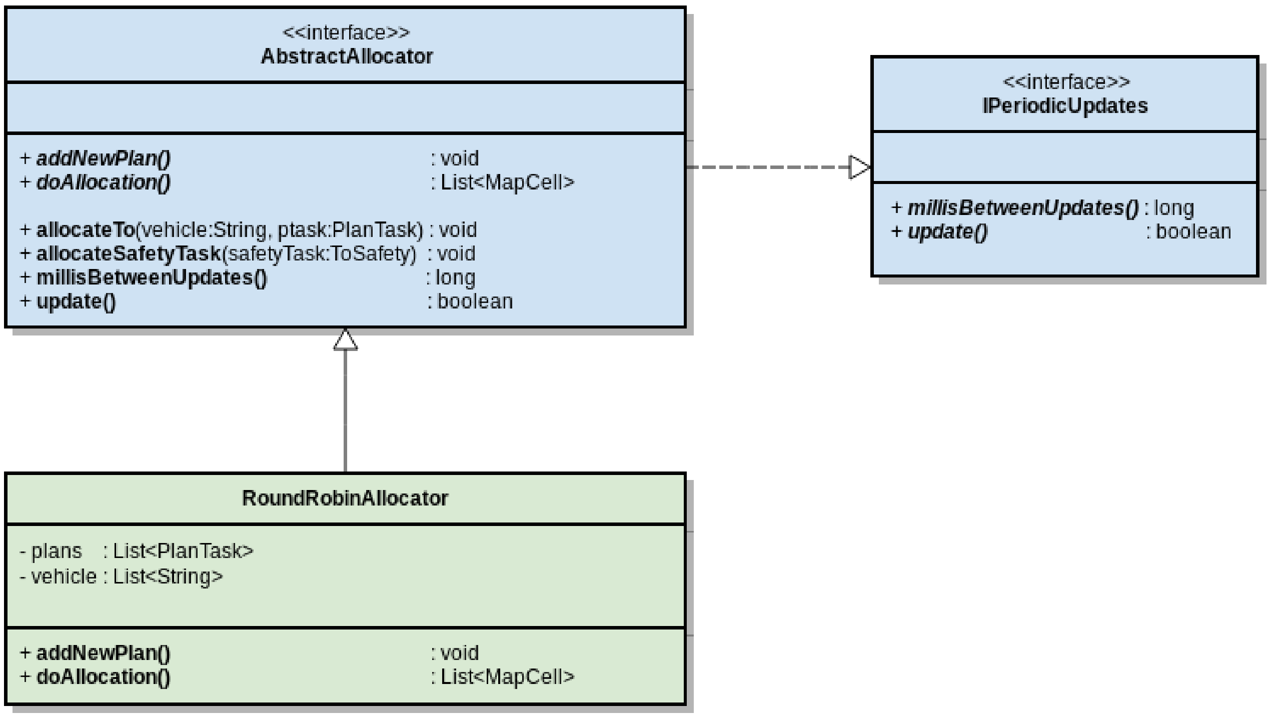
Further developments would be needed to couple *Europa* into *MvPlanning* so that planning problems and *PlanTask*'s could be described in the *NDDL* domain language, and then used by the planner.

Appendices

Appendix A

MvPlanning

A.1 AbstractAllocator



<<interface>>
AbstractAllocator
+ addNewPlan() : void
+ doAllocation() : List<MapCell>
+ allocateTo(vehicle:String, ptask:PlanTask) : void
+ allocateSafetyTask(safetyTask:ToSafety) : void
+ millisBetweenUpdates() : long
+ update() : boolean
<<interface>>
IPeriodicUpdates
+ millisBetweenUpdates() : long
+ update() : boolean
RoundRobinAllocator
- plans : List<PlanTask>
- vehicle : List<String>
+ addNewPlan() : void
+ doAllocation() : List<MapCell>

Figure A.1: UML diagram of AbstractAllocator

A.2 PlanTask

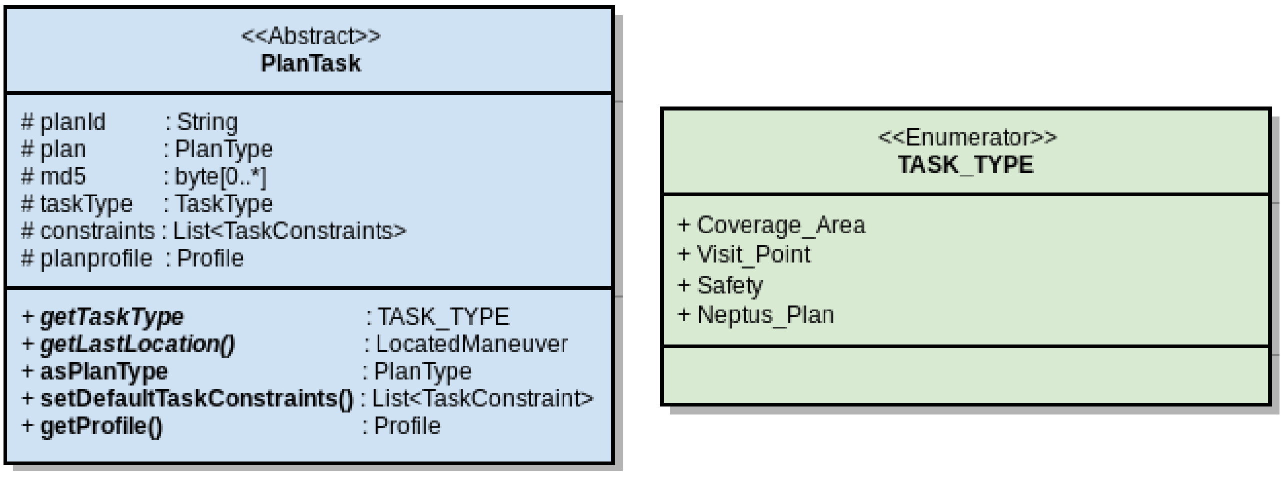
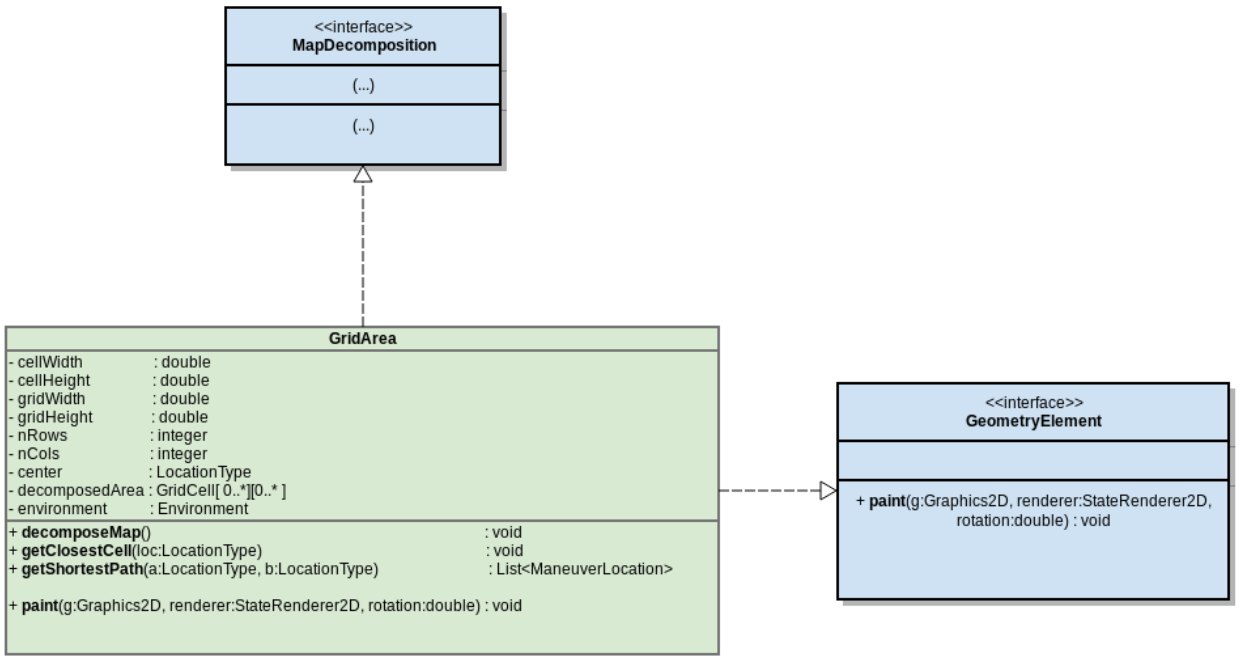


Figure A.2: UML PlanTask

A.3 GridArea



<<interface>>
MapDecomposition
(...)
(...)
GridArea
- cellWidth : double
- cellHeight : double
- gridWidth : double
- gridHeight : double
- nRows : integer
- nCols : integer
- center : LocationType
- decomposedArea : GridCell[0..*][0..*]
- environment : Environment
+ decomposeMap() : void
+ getClosestCell(loc:LocationType) : void
+ getShortestPath(a:LocationType, b:LocationType) : List<ManeuverLocation>
+ paint(g:Graphics2D, renderer:StateRenderer2D, rotation:double) : void
<<interface>>
GeometryElement
+ paint(g:Graphics2D, renderer:StateRenderer2D, rotation:double) : void

Figure A.3: UML diagram of GridArea

References

[1] G. A. Bekey, *Autonomous Robots - From Biological Inspiration to Implementation and Control.*

[2] J. L. Bresina, A. K. Jónsson, P. H. Morris, and K. Rajan, "Activity Planning for the Mars Exploration Rovers," pp. 1–10, 2005.

[3] J. Bresina, A. Jońsson, P. Morris, and K. Rajan, "Mixed-initiative activity planning for Mars Rovers," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1709–1710, 2005.

[4] M. Cummings, J. Marquez, and N. Roy, "Human-automated path planning optimization and decision support," *International Journal of Human-Computer Studies*, vol. 70, no. 2, pp. 116–128, 2012.

[5] Chen, T.L. and Prichett, A. R., "Development and evaluation of a cockpit decision-aid for emergency trajectory generation," *Journal of Aircraft.*

[6] K. Johnson, L. Ren, J. Kuchar, and C. Oman, "Interaction of automation and time pressure in a route replanning task," *Proceeding of the International Conference on Human-Computer Interaction in Aeronautics (HCI-Aero*, 2002.

[7] C. Layton and M. C. Smith, P.J, "Design of a cooperative problem-solving system for en-route flight planning-ane empirical evaluation," *Human Factors*, vol. 36(1), pp. 96–116.

[8] P. Sousa Dias, S. Loureiro Fraga, R. M.F. Gomes, G. M.Goncalves, F. Lobo Pereira, J. Pinto, and J. Borges Sousa, "Neptus - A framework to support Multiple Vehicle Operation," 206.

[9] J. Pinto, P. Sousa Dias, R. Goncalves, and E. Marques, "Neptus - A framework to support the mission life cycle," 2006.

[10] L. Chrpa, J. Pinto, M. A. Ribeiro, F. Py, J. a. Sousa, and K. Rajan, "On Mixed-Initiative Planning and Control for Autonomous Underwater Vehicles On Mixed-Initiative Planning and Control for Autonomous Underwater," no. October, 2015.

[11] F. Py, J. Pinto, M. A. Silva, T. Arne Johansen, J. Sousa, and K. Rajan, "On Mixed-initiative Coordination for Oceanographic Experiments,"

[12] "IMC v5.4.6 Specification."

[13] R. Martins, P. S. Dias, E. Marques, J. Pinto, J. B. Sousa, and F. L. Pereira, "IMC: A communication protocol for networked vehicles and sensors," *OCEANS - Aberdeen, Scotland*, pp. 1–6, 2009.

[14] S. M. LaValle, *Planning Algorithms*, vol. 2006. 2006.

[15] T. L. Mccluskey, "PDDL : A Language with a Purpose ?," *Computing*, no. 1.

[16] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[17] W. Cushing, S. Kambhampati, K. Talamadupula, D. S. Weld, and Mausam, "Evaluating Temporal Planning Domains," *International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 1–8, 2007.

[18] W. Cushing, S. Kambhampati, and D. S. Weld, "When is Temporal Planning Really Temporal ?," 2006.

[19] H. Choset, "Coverage for robotics - A survey of recent results," *Annals of Mathematics and Artificial Intelligence*, vol. 31, pp. 113–126, 2001.

[20] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258–1276, 2013.

[21] E. Galceran, "Coverage Path Planning for Autonomous Underwater Vehicles," 2014.

[22] E. Galceran and M. Carreras, "Planning coverage paths on bathymetric maps for in-detail inspection of the ocean floor," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 4159–4164, 2013.

[23] R. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36(6), pp. 1389–1401.

[24] P. Blondel, *The Handbook of Sidescan Sonar*. 2010.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*.

[26] E. R.B. Marques, M. Ribeiro, J. Pinto, J. B. Sousa, and F. Martins, "NVL: a coordination language for unmanned vehicle networks,"

[27] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, and D. Smith, "EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization,"