FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mixed-Initiative Planning of Networked Vehicle Systems

José Pinto

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Borges de Sousa

February 17, 2017

Mixed-Initiative Planning of Networked Vehicle Systems

José Pinto

Mestrado Integrado em Engenharia Informática e Computação

February 17, 2017

Abstract

Networked Vehicle Systems consist of multiple heterogeneous robots, sensors and human operators combined together and coordinated in order to achieve one or more common objectives. Planning the behavior of these systems is still a challenge for various reasons: complexity coming from the multiple interactions, system-level behavior and network topology are intertwined and uncertainty regarding the entire system state. This tend to make it difficult for human operators to take any informed decisions or do proper planning.

This thesis addresses planning of Networked Vehicle Systems under mixed-initiative interaction. Given a network of robots with heterogeneous capabilities we allow human operators to task the entire network at once by providing system-level objectives and having the system decomposing this objective into vehicle actions according to their capabilities. Moreover, we do this while maintaining the human operators informed about what the different parts of the system are doing and intervene if necessary.

For this we have developed and field-tested three separate approaches. First, we have developed a centralized planning architecture where a single planner is used to generate all low-level actions to be executed by the robots. Second, we have improved onboard autonomy by integrating a deliberative planning and execution engine onboard the vehicles and, third, we have developed a novel distributed planning framework where a centralized planner generates high-level objectives for robots deployed remotely and synchronizes its internal state with information received over fallible communication links.

Acknowledgements

This work would not have been possible without the help of my coleagues from LSTS-FEUP. Not only they were always eager to test new developments which usually require complex hardware and operational setups but they often do it under harsh conditions and for that I am very thankful.

Special thanks go to João Sousa (my advisor), Kanna Rajan, Frederic Py and Lukas Chrpa with whom I've been collaborating closely towards many of the results presented in this thesis.

José Pinto

"If a machine is expected to be infallible, it cannot also be intelligent."

Alan Turing, 1947

Contents

List of Figures

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
ASV	Autonomous Surface Vehicle
COTS	Commercial Off The Shelf
DUNE	DUNE Uniform Navigation Environment
FEUP	Faculdade de Engenharia da Universidade do Porto
IDE	Integrated Development Environment
I/O	Input/Output
IMC	Inter-Module Communications
LOC	Lines Of Code
LSTS	Laboratório de Sistemas e Tecnologia Subaquática
POSIX	Portable Operating System Interface
MOOS	Mission Oriented Operating Suite
NVS	Networked Vehicle System
RISC	Reduced Instruction Set Computer
ROS	Robot Operating System
ROV	Remotely Operated Vehicle
SDK	Software Development Kit
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language for Transformation
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
VTK	The Visualization Toolkit
WWW	*World Wide Web*

Chapter 1

Introduction

Recent advances in computing and micro-electronics have led to the development of numerous low cost robots that can be used by humans for dummy, dull and dangerous operations. These robots can be used not only at home but also in remote and hostile environments which are unfeasible or unsafe for humans.

Moreover, multiple robots can be combined together and coordinated to achieve common objectives as defined by the human operators. Tight coordination, where robots are controlled from a system-level perspective and their individual capabilities, configurations and limitations are abstracted away from the operators, will allow for a new class of applications that are about to become reality. We call such networks a *Networked Vehicle System* and an example can be seen in Figure 1.1.

As it can be seen on the figure 1.1, NVS can be composed not only by robotic vehicles but also manned assets (such as ships, cars), uncontrollable devices such as communications satellites and drifting sensors and static assets such as base stations and communication gateways. Communications is an important part of these systems (and thus the Network in its name). Robots connect to

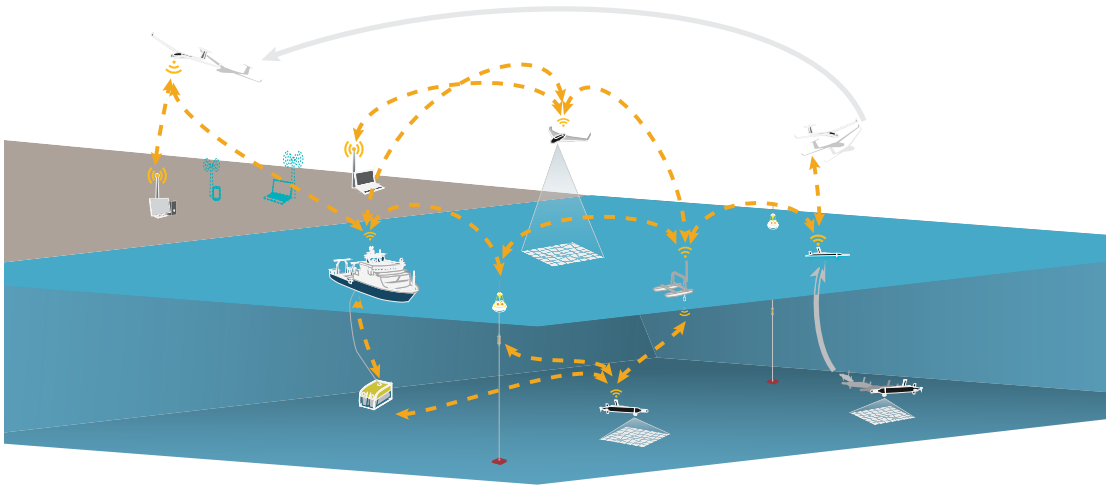

Figure 1.1: Networked Vehicle System deployment example

the network but they are also network nodes, which means they can relay and forward data to and from their peers.

Since robots are capable of moving and possess one or more communication means, they can also be used to transport information from different locations in the world (commonly called as "data mules"). So, not only the geometry of the network defines if and how the nodes communicate with one another but also the currently defined behavior defines how the nodes will be able to coordinate their actions and communicate the results back to where they were requested.

Human operators typically request objectives from the system and they are the ultimate decision makers. This happens because operators can use all past experience to answer in cases such as large-scale hardware malfunctions, enemy behavior or other unforeseen situations. Thus, it is important to keep human operators in the planning loop: they should be aware of what are the current and future global states of the system, supervise automated planning and possibly change it.

Planning the behavior of an NVS then consists in defining the behaviors to be executed by the controllable devices (robots) while taking into account their capabilities and limitations and coordinating the communications of the entire network. Moreover, users should be part of planning loop as they can provide precious information and knowledge that can not be part of any mathematical domain model.

1.1 Motivating Scenarios

Despite the huge possibilities of these technologies, Networked Vehicle Systems tend to be so complex that is difficult for human operators to understand the current state of a system composed by multiple mobile parts with different constraints and objectives which makes it even harder to grasp the real conjoint capabilities of such systems.

1.1.1 Scientific Applications

There exist multiple scientific applications for these systems. There are many disciplines that require synchronized observations of the environment for effects of comparison and/or measuring the evolution of some property.

More specifically, oceanography typically requires associating remote sensing with in-situ observations. A similar approach can be used to associate observations from air with others taken at the surface and underwater by different types of vehicles. This is only possible by coordinating multiple asset types and since there will be interest in monitoring storms and remote locations, unmanned assets should be employed for safety.

1.1.2 Emergency Response

In catastrophic events such as earthquakes, tsunamis, ship wrecks and other large-scale emergency situations, a quickly deployable network of robots can greatly aid first responders in the

field. These systems can be used to establish contingency communications even if all networking infrastructure is gone. Moreover, a group of aerial robots can provide a global picture of the area and actively search for survivors. The way the assets is deployed and used, however, needs to be coordinated in order for the assets to do search and maintain connectivity with base stations at the same time.

1.1.3 Agriculture

Agriculture applications typically require robots to cover large areas of land for instance to create a map of the soil and/or plantation conditions. A lot of companies is now pursuing this application using aerial vehicles (drones) to survey ares and produce picture mosaics using visible light, infrared and hyper-spectral cameras. Existing companies tend, however, to use a single vehicle to cover large areas which often requires multiple deployments due to limited flying autonomy.

Using a multi-robot approach the survey could be executed faster and automatically taking into account flying / landing constraints such as those required by law. Automated planning could thus streamline the operation of drones for agriculture surveys.

1.1.4 Industrial

Industrial robotics is already very much established and one of the reasons the prices of COTS drones has come down so substantially. Industrial applications typically rely on an underlying communications structure and very restricted robot mobility. However it is our opinion that NVS can also be applied to industrial applications in factories with rapid-changing production requirements. In those scenarios, generic robots can be adapted to different production schemes being used for transportation, assembly and packaging, for instance.

1.1.5 Military

There are numerous applications for Networked Vehicle Systems in the military. That is because NVS functions in a similar fashion to that of an army in enemy territory: it has to adapt to areas with no underlying structure and reorganize teams according to changing objectives.

Unmanned Aerial Drones are already being used extensively for intelligence gathering and directed attacks. However, current day missions require human pilots controlling almost every move of the remotely deployed robots. NVS approach would allow lower cost devices that can, for instance, be deployed for longer periods of time and become active when required like when alarms are triggered or when new objectives are requested by the remote operators.

1.2 Problem Statement

This dissertation addresses the planning and coordination of Networked Vehicle Systems. In these systems, different vehicles possess different sensors and actuators, allowing them to perceive different parameters of the environment and move according to their specific capabilities. Moreover,

relative vehicle poses will define the set of available communication links that are created and destroyed according to their movements and respective communication means.

Not only the fact that sensors and actuators have associated errors but also the fact that used communication links are fallible and change according to the behaviors executed by the robots, increase the difficulty in the process of planning. Planners must not only account for the vehicle capabilities and how they can be used to attain desired objectives but also they must plan how the commands and results are communicated to, from and between the robots. Moreover, after a plan is issued to a network of robots, its execution may not go entirely as predicted. Plans should be adapted onboard or revised centrally in order to continue pursuing the original objectives.

Given a Networked Vehicle System, we want to address the problem of decomposing a set of user-specified high-level goals into a set of behaviors to be executed by the vehicles that, under reasonable assumptions, achieve the desired objectives. This problem can be further decomposed into 4 sub-problems.

First, we need to create planning domain models that comprises heterogeneous vehicle capabilities and potential failures in communication, sensing and actuation. *Second*, we need to plan the behavior of individual vehicles according to a set of user-specified goals, individual vehicle capabilities and estimated world state. *Third*, we need to provide graphical interfaces that allow users to perceive the current and future estimated state of the system and supervise the planning loop, being capable of intervene in case of unforeseen situations. And *fourth*, these developments must be integrated with existing tools in order to be used in real-word deployments.

1.3 Thesis Outline

In chapter 2, we overview robotic frameworks starting from its origins till the ones most used nowadays. After a comparison, we describe in more detail the LSTS software toolchain that was used as the basis of this work.

In chapter 3, we describe 3 different approaches that were tested for mixed-initiative planning of autonomous vehicles: Centralized Planning (3.1), Onboard Deliberation (3.2) and Distributed Deliberation (3.3).

Chapter 4 we document 3 different field experiments that were undertook to validate all the 3 different approaches, respectively.

Chapter 5 concludes this thesis with an overview of the main contributions as well as discussion of results and guidelines for future work.

Chapter 2

Related Work

In this chapter we refer existing work aiming at defining robot behavior and/or coordinating the actions of multiple robots. At the Underwater Systems and Technology Laboratory several related technologies have also been developed that are used for the implementation of this thesis. As such, they are also listed in this chapter.

2.1 Robotics Frameworks

Robotics software typically comprise multiple layers of abstraction, from low level control and actuation to high-level knowledge acquisition and decision making. As such, several software frameworks have been developed by industry, academia, hobbyists and military to simplify common tasks, hide unnecessary details and support modular development of robotic applications. In this section we give a brief historical background on the development of robotic frameworks and discuss the state-of-the-art in frameworks which are aimed towards the support of interoperable multi-robot systems and are agnostic of both end application and robot types.

2.1.1 Historical Background

In the early years, most robotic software was developed together with the hardware and was not meant to be reused. This was the case for most robots developed in the 20th century. With the advent of commercial robotics (Sony AIBO, Lego Mindstorms) extra care was given to allow the development of robot applications by end-users. Thus, the first application-agnostic but robot-specific software frameworks were born: Open-R SDK for AIBO and Robolab for the Lego Mindstorms.

Robolab, targeting educational robotics is a visual programming language developed on top of LabView ([ECR00]). Robolab allows the definition of flow diagrams that specify how inputs from sensors are transformed into actuation outputs. An example robotic application for Lego Mindstorms can be seen in figure 2.1.

In May 1999, Sony introduced the AIBO “robotic pet". These robots were developed targeting domestic use and entertainment. However, hobbyists and academia soon realized that such devices

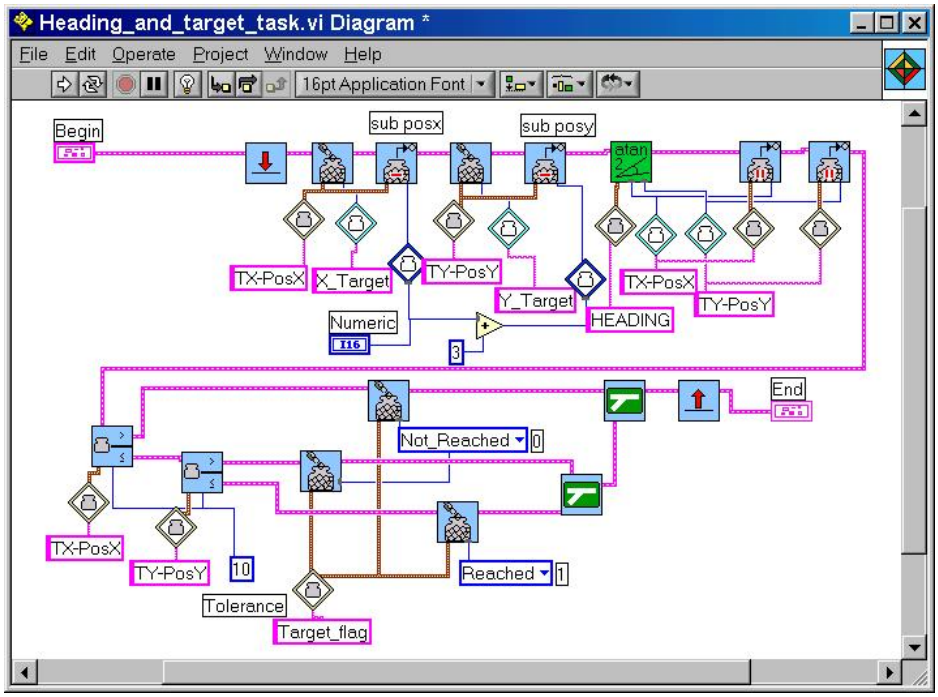

Figure 2.1: Robolab application for the Lego Mindstorms

would be perfect for experimentation considering their low cost and reliability ([SB03]). Despite initially Sony not providing any SDK for extending the robot functionalities and even preventing others from doing so ([NYT01]), later in 2002 Sony released a full set of tools for developing new applications and behaviors for their robotic dogs, called "AIBO Software Development Environment" which included, among other tools the Open-R SDK.

Open-R SDK is a C++ software framework that allows the development of native applications for the AIBO. Instead of developing applications that handle all sensor inputs and compute the outputs to send to actuators in a single loop, this SDK requires the applications to be built in a modular fashion. Programs are decomposed into "Open-R Objects" which react to incoming messages and are allowed to send messages to other objects. Messages flow between Open-R objects through "channels" which are typed in the sense that only a specific type (field structure) of message is allowed.

In order to interact with the hardware, 2 objects are provided by Sony that react to incoming messages by changing the hardware state: *OVirtualRobotComm* and *OVirtualRobotAudioComm* that allow accessing / changing the state of the robot joints and sounds, respectively (see example in Figure 2.2). This clear separation of concerns and typed interaction mechanism is still used today in the most important robotic frameworks.

2.1.2 The Player Project

The Player project started by being an umbrella project encompassing visualization software (Stage, and later Gazeebo) and an Hardware Abstraction Layer (HAL) for robotic devices (Player).

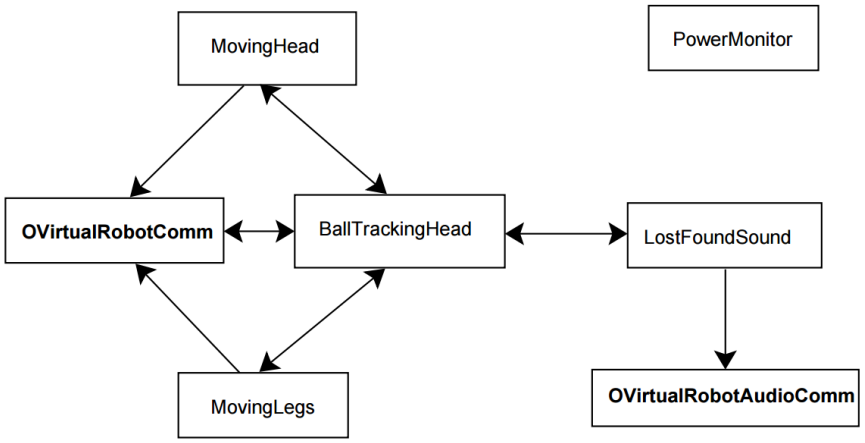

Figure 2.2: Component Diagram for an Open-R SDK application

Player differs from other robotic frameworks in the sense that it doesn't provide any control architecture but only hardware abstractions much like an Operating System provides common abstractions for Input/Output devices.

Also, Player is not language-specific as it allows user applications (developed in any language) to interact with robots using a client-server architecture where the robot (server) accepts connections to its devices (abstracted as character streams over TCP).

The Player project deals with a single problem and solves it clearly: robotic hardware abstraction. The Player architecture is generic enough that allows common (API) abstractions for different robot hardware. Moreover, the fact that a generic visualization application exists (Stage, later Gazeebo) also contributed to its success. Player/Stage was one of the first Open Source robotic frameworks and is still in use today by an active community (mostly academia).

2.1.3 OROCOS – Open robot control software

Introduced in 2001, OROCOS had the ambitious goal of becoming "the general-purpose and open robot control software package" ([Bru01]). Instead of reinventing the wheel with new communication protocols and standards, it emphasized reuse of already existing tools and aimed mostly creating a centralized repository where different developers can contribute compatible software. More specifically, OROCOS embraced the use of the following technologies and related libraries:

CORBA Considered at the time the *de facto* standard for inter-process communications. Uses IDL files for the definition of communication protocols;

RTLinux Hard real-time microkernel that allows running a full Linux operating system as a preemptive process;

XML Used for all configuration and data files;

DocBook Used for the project documentation;

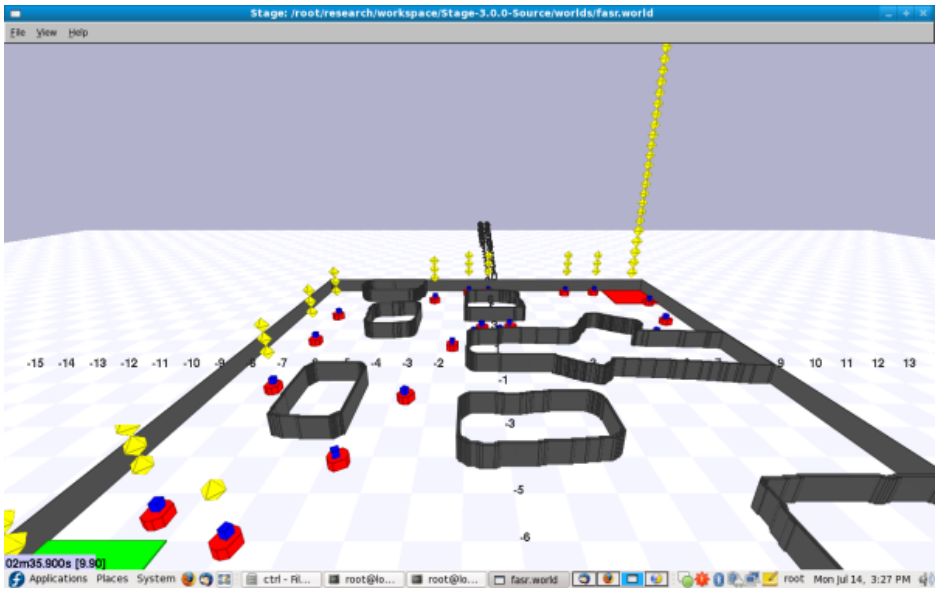

Figure 2.3: Screenshot of Stage visualizing data from a Player server

VRML for 3D visualizations;

Modelica used for physical simulations.

Like OROCOS, other roboticists opted for using the CORBA communications standard for inter-module communications. This allows clear separation of concerns between implementation and end-users of the APIs as only the interface is shared using IDL (Interface Description Language).

2.1.4 The MOOS – Mission Orientated Operating Suite

MOOS is a project initially developed at Oxford University as a set of libraries and applications designed to facilitate research in the mobile robotic domain ([New08]) and including support for communications and navigation. The main idea behind the design of MOOS is that robotics code should be modular: different authors create different applications / processes and rely on a robust communications library to share data between applications. As such, a "MOOS community" is a set of applications (called *MOOSApps*) configured to run together by sharing data with each other.

At the core of every MOOS community there is the *MOOSDB* application. This application accepts (TCP/IP) connections from other modules and allows publishing and subscribing of data. The data is stored in a key-value database which allows only two types of variables: Text and floating point numbers. The limitation on the number of data types may be overcome by using formatted text variables. For instance, it is quite common in a MOOS community to pass variables that contain comma-separated values serialized as text as a mean to serialize structured data.

Similarly to what happens with the Player project, the nodes don't need to reside in a single computer as they communicate with the MOOSDB over TCP/IP. The communications library is

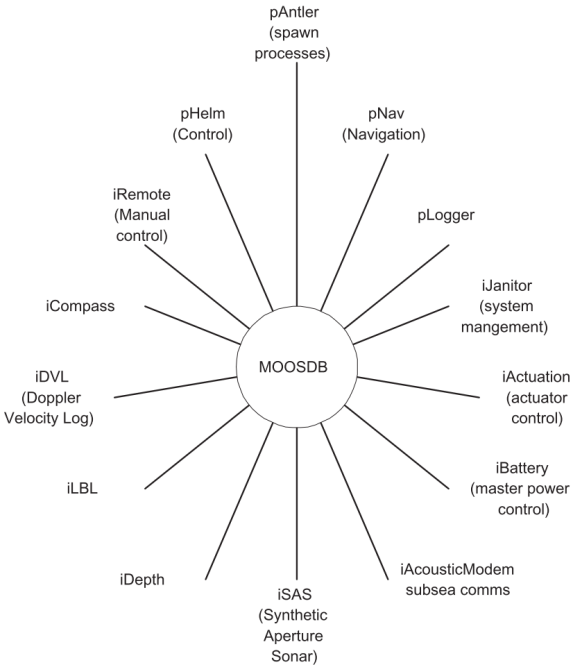

Figure 2.4: Components of an AUV MOOS community.

lightweight and the MOOS core components have no software dependencies which makes it very easy to compile and distribute MOOS to different CPU architectures and operating systems.

All nodes in a MOOS community can be uniquely identified by its name and all messages are tagged with its source which is important for later revision and debugging. Moreover, in newer versions of MOOS, all messages are also tagged with originating MOOS community, allowing messages from different communities to be exchanged. This typically involves having a special node in the communities that connects to more than one MOOSDB applications and selectively forwards information between them.

Moreover, MOOS provides a set of tools and libraries that are reused across most applications called the *MOOSGenLib*. It includes:

- Platform-independent serial ports.
- Thread safe configuration reading.
- String manipulation/parsing tools.
- Geodesy tools.

The configuration reading tool specifically provides a generic mechanism for applications to read their parameters from one or more files called “mission files”. A single mission file may be read by more than one applications because the allowed file syntax allows specifying which application will read each configuration block. For instance, in order to bootstrap a complete MOOS community (multiple processes), a mission file can be passed to a *pAntler MOOSApp* that

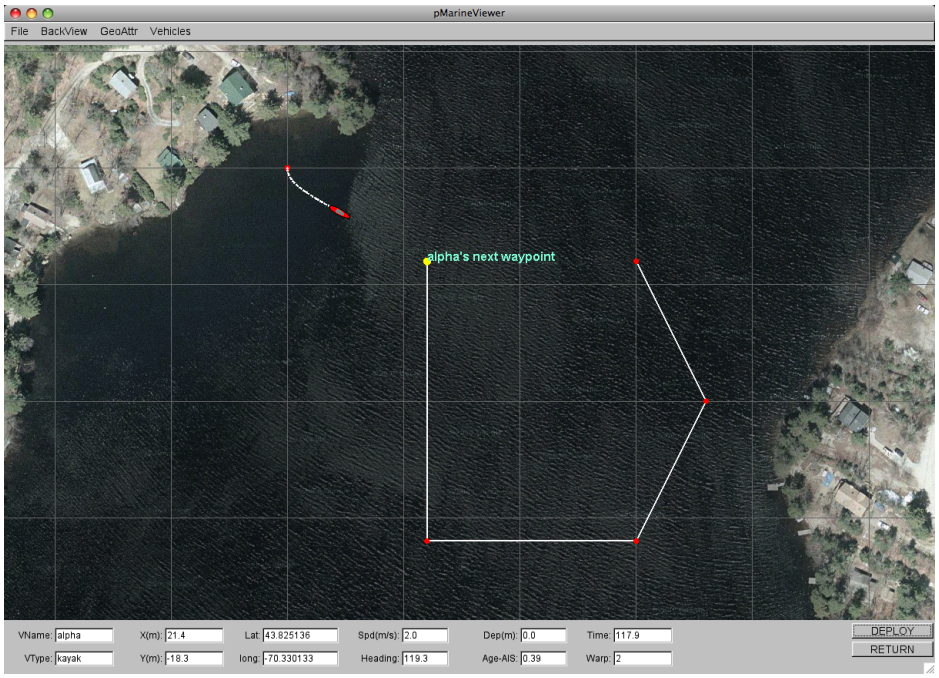

Figure 2.5: *pMarineViewer*: MOOS application for real-time data visualization

will start any processes specified in the configuration file and also pass this file to the processes as their configuration. This is currently the standard (recommended) way a MOOS community is launched.

From its start MOOS had a lot of adoption from marine robotics research groups and, as time advanced, it grown to be a collection of libraries to which these groups contributed. As a result, this software is still in use mostly by marine robotics groups. Supported robots include AUVs of all sizes, ASVs, ROVs and even UAVs (however, not including the low-level control part).

2.1.5 ROS – Robot Operating System

ROS is a collection of software libraries to aid in the development of robotic applications (much like MOOS and Player). It evolved from a framework that was used to support the Stanford AI Robot (STAIR), originally called the *switchyard* project, back in 2007. From 2008 until 2013, ROS was developed primarily by the engineers at Willow Garage – a robotics group that targeted the development of open source robotics and profit from selling compatible hardware and tools support. Willow Garage ceased to exist in 2013 but a huge community of roboticists continued its development and adapted it for supporting all kinds of robots and hardware systems.

Despite being called an Operating System, "ROS is not an operating system in the traditional sense of process management and scheduling; rather it provides a structured communications layer" on top of the host operating system. The authors state the philosophical goals of ROS as being *Peer-to-peer*, *Tools-based*, *Multi-lingual*, *Thin*, *Free* and *Open-Source* ([QCG+09]).

Figure 2.6: Turtlebot (left) and PR2 (right) – two flagship ROS robots, develop by Willow Garage

ROS (similarly to the other previously mentioned projects) relies on a microkernel design, where multiple processes are networked together to support a single robotic system. The entire robotic system, however, relies on a central node service for node registration and look-up, called *name service* or simply *master* node.

The ROS specification resides at the messaging layer and, as such, different compatible implementations can be developed and put to work together which allows for language-independent development. The list of ROS-supported languages include C++, Python, Java and LISP. Most low-level drivers are developed in C++ but most ROS-based projects also contain a large extent of processes and tools developed in Python, which is the second most common language for ROS. The reason may be related to the fact that most ROS tutorials use this language for its convenience and conciseness.

ROS re-uses code from other open-source projects like Player (hardware drivers, navigation, simulation), OpenCV (vision), OpenRAVE (planning) and PCL (3D point clouds), among many others. All these projects are wrapped into ROS packages, name given to a collection of tools / libraries revolving around some API / hardware device. ROS also supports automatic update of ROS packages and dependency management, resembling what could be a Linux distribution targetting robotics research.

In terms of communications, ROS supports both asynchronous publish-subscribe communications and synchronous remote procedure calls. All communications are done through typed messages whose structures must be known to both communicating ends before transmission. When a node publishes to a topic, either a set of nodes that subscribed to that topic will receive it or a single node that provides the named service will receive the message. The centralized name server explicitly ensures that no two nodes provide the same named service, however when a message is received no information about the source is included (unless a field for that information is

Module	Discription
`rxbag`	Replay a ROSBag file to the local network
`rxconsole`	Allows seeing all debug (text) information coming out of different ROS nodes
`rxgraph`	Visualization of the ROS communications graph
`rxplot`	Plot scalar variables that are updated through messages
`rviz`	3D visualizations of robot poses / navigation

Table 2.1: Different tools provided by ROS for data visualization

explicitly added to the message by the application).

In ROS, message structures are specified in text files that can be used to generete code in different languages. As a result, different research groups can contribute to a single set of message specifications and afterwards use this as the shared communication protocol. However, no partner is restricted to use only those messages, as multiple message specification files can be used at a single time.

ROS message structures can be logged to files (called *ROSBag*'s) and replayed in a modified network graph (with a different set of running nodes). This can be extremely useful when the users need to isolate a node for testing / debugging. These flexible replay mechanisms are important in robotics research because experiments typically require very specific setups that can be expensive or complex to reproduce. Currently there exist numerous tools that can open ROSBag files and replay / visualize its contents such as the ones in table 2.1.

At the time of writing ROS is being used by several robotics research groups as well as bundled together with several comercially off-the-shelf robots like the TurtleBot 2 (Clearpath Robotics) or the Sparus II AUV (Universitat de Girona).

2.1.6 The LSTS Toolchain for Networked Vehicle Systems

The LSTS Toolchain (initially developed by Laboratório de Sistemas e Tecnologia Subaquática from Porto University) is a set of open source tools aiming the support of Networked Vehicle Systems [dSeTS09b]. In early 2004, the original idea was to devote operators with tools required to do visual planning and simulation for heterogeneous vehicles (AUVs and ROVs). The objectives quickly evolved into being a way of supporting the operation of ing thevehicle fleets during the different operation phases (planning, simulation, execution, data revision).

The approach for the development of LSTS Toolchain differs from that of ROS or MOOS in the sense that, purposely, the authors wanted to take control of all the different control layers, from low-level sensing and actuation to high-level planning and data fusion. So not only low-level hardware abstractions and communication mechanisms were devised but also a biased view and implementation of the control architecture and vehicle inter-operability.

Despite the fact that a single institution was in charge of the early development, the LSTS tools have always targetted generic, robot-independent usage. As such, LSTS software architecture was developed taking into account that it shall be used to support different robot and communications

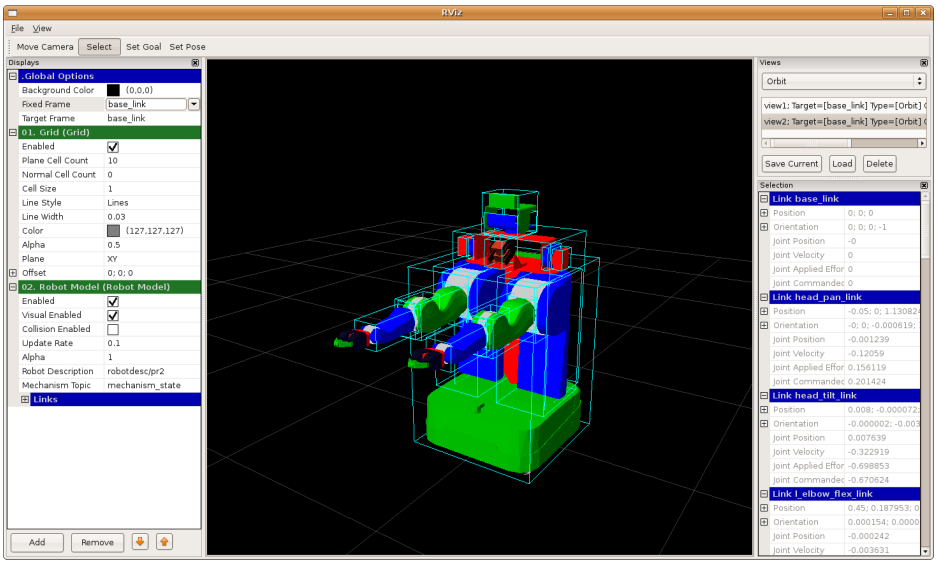

Figure 2.7: RViz application used to visualize the pose of a PR2 robot in 3D.

hardware, mission purposes and operator needs. The LSTS Toolchain is divided into two main modules: The **DUNE** embedded software (described in section 2.2) and **NEPTUS** desktop software (described in section 2.3.2). All communications inside these tools and between them is done through a common middleware, called **IMC** which is also described in section 2.3.

2.1.7 Discussion and comparison

The main objectives of using a software framework is to be able to reuse existing software and create modular software that can be reused for uses not foreseen at the time of writing the code. From this perspective, all presented frameworks provide a set of reusable components that target different robotic applications and use different approaches towards modularity of components. Player/Stage and OROCOS use a 3-tier architecture where logic, visualizations and data storage are clearly separated. In the case of MOOS, ROS and LSTS, these concerns are spread between different components that all communicate with one another using a common communication system. Despite maybe more difficult to start with, this flexibility can be useful for applications that do not follow a standard client-server communication scheme as is the case of ad-hoc vehicle networks.

When considering multi-robot applications, all the frameworks can be used to connect multiple robots but only the LSTS toolchain specifically targets such usage. The main difference resides on the fact that **IMC** does not require reliable communications and that peers are discovered dynamically as they become connected to the network. All the other frameworks, when considering multi-robot networks, have some kind of central node that relays all communications, something that doesn't exist by design on the LSTS toolchain. The approach used by ROS and MOOS is to add a special component that listens to data from a local network (MOOS community, ROS master) and forwards it to remote peers (using TCP) – this is explained by the micro-kernel approach

of these two frameworks, as opposed to LSTS where there is a single node (process) per CPU and it is possible to use shared memory inside node modules for efficient communication.

All the presented Robotic Frameworks provide a set of utility libraries and hardware abstraction layers that help developers be more productive by focusing on the application instead of low-level interfaces. Currently ROS has the largest code base with "over 7.3M lines of code. However, only 27% of the code in ROS is released yet. There is a big portion (73%) of code in ROS we don't know much about" ([Kue13]). Nonetheless, this is currently the framework that provides the largest and broadest utilities library. MOOS bundled utilities target mostly configuration parsing, navigation, path planning and geodesy and is the smallest framework in terms of LOC. LSTS Toolchain is divided between desktop and embedded software, on the **DUNE** (embedded) side, there are different math (angles, matrices, geodesy, vectors, ...), configuration parsing, navigation, compression, networking and I/O utilities.

In recent years multiple efforts have been made to create robotic frameworks from scratch which hints that there is no silver bullet that covers all possible applications and scenarios. As a result the robotics community is fragmented among different code bases and approaches. With such a number of diverse open source frameworks, certainly it would be beneficial to merge some utilities and bring communities together. For the development of this thesis, we found the LSTS toolchain to be the most adequate. We next list some of the reasons:

Multi-robot support The decentralization of communications and service discovery features makes it possible to have ad-hoc networks of robots;

Desktop tooling support As opposed to most other frameworks, the LSTS toolchain includes a large library of visual widgets and libraries that are tightly integrated into **NEPTUS**;

Lightweight onboard software The fact that the **DUNE** onboard software uses a single process and shared memory for inter-thread communications allows for a lower memory and CPU footprint. Something necessary for multi-robot simulations and low-cost (computationally limited) robots.

2.2 DUNE: DUNE Uniform Navigation Environment

The **DUNE** (**DUNE** Uniform Navigation Environment) is an Open Source software, originally developed by LSTS to target cpu- and memory-limited enviroments such as those available in networked sensors and autonomous vehicles like AUVs, ASVs or UAVs. **DUNE** is developed in C++, targets POSIX environments and has no library dependencies except for the standard C++ library (*stdlib*). This makes it very easy to port **DUNE** to virtually any CPU architecture running a POSIX-compatible operating system. For example, **DUNE** has been tested to run in desktop / laptop computers running OSX, Linux, Windows and Solaris, as well as embedded devices running some flavor of Linux like the Geode LX900, the Raspberry Pi, the BeagleBone Black, among many others.

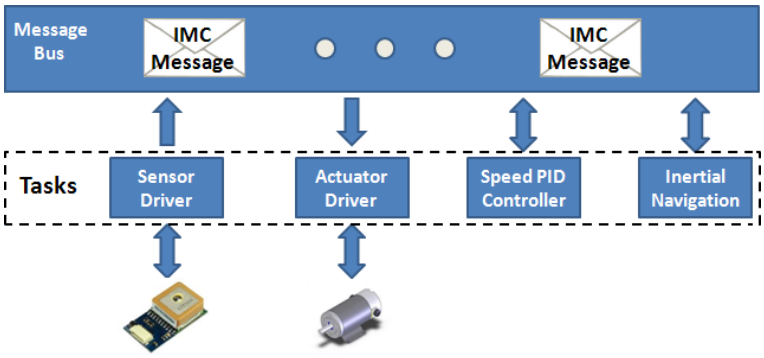

Figure 2.8: Abstract representation of **DUNE** message bus

Just like ROS, MOOS or Player/Stage, **DUNE** uses a publish/subscribe system for message-passing between modules that run concurrently in a local system or distributed in the network. A major difference between **DUNE** and the other infrastructures, however, is that each **DUNE** instance consists of a monolythic process holding multiple components (called Tasks) each running on its own thread. To prevent Tasks running on the same **DUNE** instance from blocking one another, **DUNE** uses asynchronous message passing for communication between Tasks. This approach, while apparently limitative, is the reason why **DUNE** is virtually safe from memory corruption and deadlocks while still being light enough to run in systems with limited memory and CPU power.

2.2.1 Architecture

The **DUNE** software is divided in modules called Tasks which have a (possibly changing) user-provided configuration and communicate with one another following a publish/subscribe pattern. As such, tasks do not interact directly with each other but can only send messages to the other tasks. All message passing is done asynchronously: posting is done in a different thread from reception.

When initialized, each task registers itself as subscriber of topics and also acquires at least one entity identifier that will be used to tag all of its published messages. Moreover these are also the identifiers used to address any specific task running inside a **DUNE** system.

All tasks extend the class `Tasks::Task` (see Figure 2.9) that provides empty implementations for responding to life-cycle events and also contains method for sub-classes to subscribe to topics. In **DUNE** , topics correspond to **IMC** message types. This option was taken so that recipients can use handlers for specific structures (types). Registering to topics is done by using C++ templates as shown in Figure 2.10.

The base class `Tasks::Task` has a reference to a `Tasks::Context` singleton instance. This shared object is used so that all tasks are able to communicate and address other tasks in the local **DUNE** system. Communication is done via the `IMC::Bus` object with which all tasks subscribe as listeners and publish messages to. When a message is published, it gets cloned (deep copy) and posted to the inbox of its subscribed listeners.

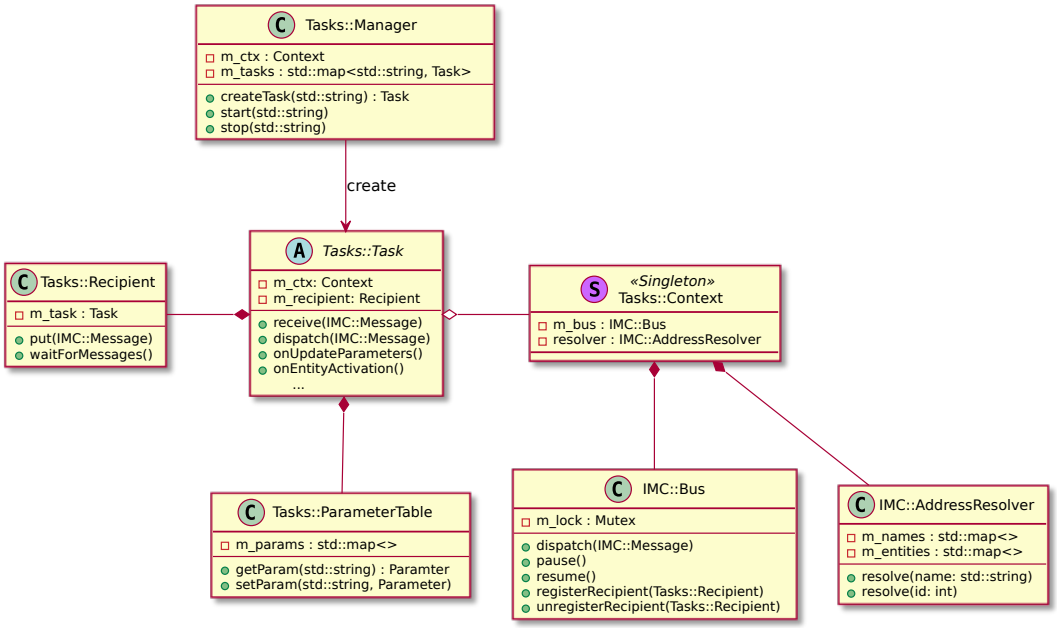

Figure 2.9: Simplified UML class diagram for **DUNE** Task

```
1 //! Bind a message to a consumer method.
2 //! @param task_obj consumer task.
3 //! @param consumer consumer method.
4 template <typename M, typename T>
5 void
6 bind(T* task_obj, void (T::* consumer)(const M*) = &T::consume)
7 {
8   bind(M::getIdStatic(), new Consumer<T, M>(*task_obj, consumer));
9 }
```

Figure 2.10: Template function used to subscribe to typed messages.

Inboxes are implemented as `Tasks::Recipient` objects that simply act as a queue of incoming messages that can be posted to and polled by the listener object. The result of this architecture is that message posting uses the thread of the message publisher but message reception is done on a separate thread, controlled by the listener. This is a practical implementation of asynchronous communication between the tasks.

2.2.2 DUNE Task life-cycle

Each task runs on its own Thread and reacts to a series of life-cycle events as well as incoming messages to which it has subscribed. In order to implement its life-cycle (Figure 2.11), virtual class which has default (empty) implementations of the methods invoked by the **DUNE** runtime at different initialization and shutdown phases as well as runtime parameter changes:

onResourceAcquisition(): During initialization stage and after parameters are loaded, used to reserve system resources such as file handles, sockets, etc.

onResourceInitialization(): Right after all tasks acquire needed resources, this method is invoked so that they can be initialized on all tasks.

onEntityReservation(): Called during initialization when tasks reserve entity identifiers required for normal execution (by calling the inherited method `reserveEntity()`).

onEntityResolution(): After all tasks have reserved entity identifiers, this method is called when tasks can resolve the identifiers reserved by any other task.

onRequestActivation(): Called when the task has been requested to be activated by another task. It can be used to block activation if some activation conditions cannot be honoured by calling the method `activationFailed()`.

onActivation(): Called when the task has become active either because it is configured to run at start or its activation has been requested by another task.

OnRequestDeactivation(): When the task is about to be deactivated, meaning it will stop executing in the near future. The task can block its deactivation by calling the method `deactivationFailed()`

onDeactivation(): Called when the task has become inactive.

onUpdateParameters(): Called when the parameters for the task are loaded from disk and whenever they are changed at runtime (by users).

onMain(): Called once the tasks is running.

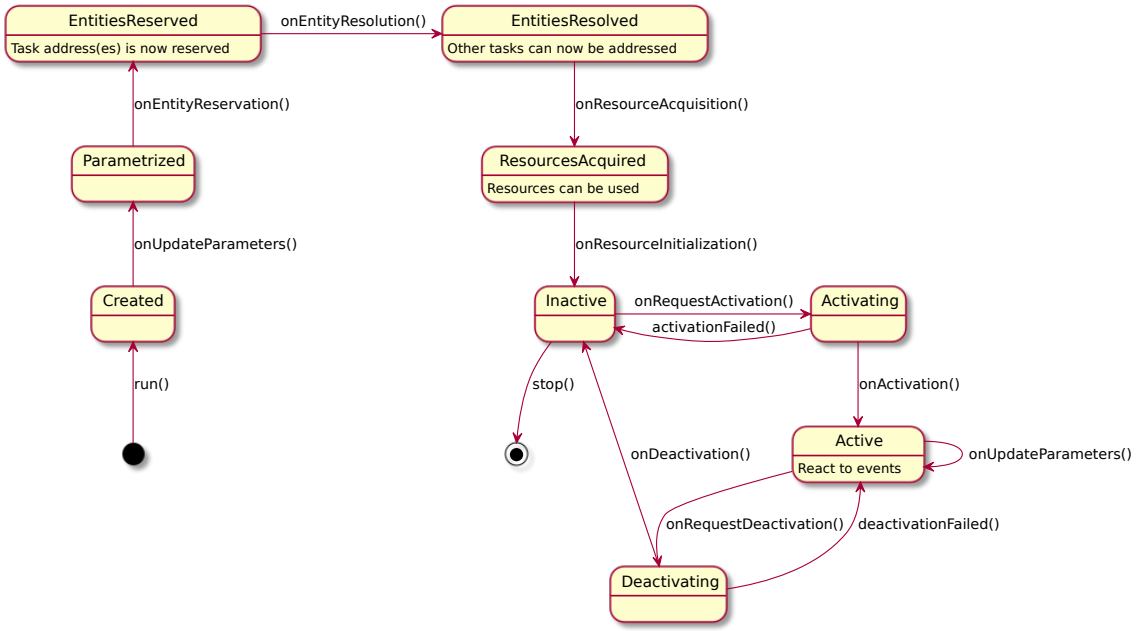

Figure 2.11: **DUNE** Task's life-cycle stages

Actuators:	Device drivers for actuators such as servo-motors, thrusters, robotics arms, etc.
Autonomy:	Plan generation and deliberation. Also includes plan adaptation to failures.
Control:	Translate between desired control references and underlying actuation.
Navigation:	Derive localization from sensor readings.
Plan:	Plan storage and execution.
Sensors:	Device drivers for all kinds of sensors.
Simulators:	Simulation replacements for various sensors and actuators.
Maneuvers:	Vehicle behaviors that involve parametrized movement.
Monitors:	Perform diagnostics but don't send any commands.
Supervisors:	Monitor execution of other tasks and potentially send commands.
Transports:	Translate messages from local bus to other formats / network protocols. Includes logging and communications.

Table 2.2: **DUNE** task types and respective namespaces

2.2.3 Task scopes and organization

DUNE is a flexible piece of software in that it supports very different types of environments, ranging from sensors, desktop simulators or robots of different kinds. As such, **DUNE** consists on a large number of tasks that can be selected and rearranged for different needs. Tasks are divided in namespaces according to their type as seen on table 2.2.

The separation made between **Monitors** and **Supervisors** is done to distinguish those tasks that can change the behavior of the system from those that just perform diagnostics. Moreover, there is usually a clear separation between sensing and actuation hardware and thus the tasks responsible for the hardware supported can be divided in the same way. The **Simulators** namespace contains software-only simulators for both actuators and sensors that can be used as a replacement for their hardware counterparts.

The **Transports** tasks are used to "transport" messages from the local bus to and from other locations / environments. For example, `Transports::Logging` task is in charge of logging all messages to disk and `Transports::Iridium` handles incoming and outgoing satellite communications.

The **Plan** namespace is used for plan execution while **Autonomy** is used for simplistic plan deliberation when critical situations are detected such as a vehicle becoming stuck underwater or not floating to the surface. Onboard autonomy is currently simplistic because **DUNE** goal is primarily to provide navigation and control for networked vehicles and sensors in support of simple plan execution. Instead, we opted to use other external autonomy tools such as **T-REX** (described later in this thesis).

2.2.4 Control and Navigation

Just like its name suggests, **DUNE** excels at providing a uniform navigation environment. What this means is that irrespectively of the sensors used to estimate a system position (localization) or actuators used to interact with the environment to achieve desired control references, there are standard structures (**IMC** messages) that can be used for higher level representations of the system state.

To provide a more specific example, consider navigation in AUVs and ASVs. Simple ASVs can use just GPS and a compass to acquire their position which can be derived to find the vehicle's velocity. All these data are encoded in the message `EstimatedState`. In AUVs, however, this simple approach is not valid. In the case of AUVs some kind of inertial sensor must be used to estimate the position while moving underwater. Typically, the same inertial sensor is used to compute the vehicle velocities and from those a position estimate is integrated but the end result is the same: an `EstimatedState` message that can be used by other tasks as the assumed real world position of the vehicle.

2.2.5 Planning and Autonomy

A **DUNE** plan consists of a `IMC::PlanSpecification` message (depicted in Figure 2.12). Tipically these plans are defined by human operators and specify a set of behaviors (maneuvers), conditions for transiting between behaviors and messages triggered upon transition. A **DUNE** system includes a `Plan::DB` task that allows storing multiple plans, identified by their name. This allows beforehand loading of mission plans and also have special plan identifiers for different possible contingencies.

2.3 IMC: Inter-Module Communications Middleware

In the early days of LSTS (beginning of 2000's), there was no standard way for the vehicles to communicate. This is understandable back then since the existing (REMUS-class) AUV didn't have wireless communications and all mission scripts and logs needed to be uploaded/downloaded via the vehicle's FTP server. The other vehicle that LSTS was operating was an ROV and in that case the vendor message protocol, working over UDP, was used to convey all real-time telemetry and commands.

That was back then. Soon after a Wi-Fi communications radio was added to the Isurus AUV, the development and specification of Seaware, a publish-subscribe middleware for autonomous vehicles [Mar06], started. Seaware later evolved to IMC (Inter-Module Communications) targeting not only aquatic vehicles but also aerial [MDM+09].

IMC is a message-oriented protocol not targeting a specific vehicle type, communication link or network topology. What this means is that all information is conveyed in the form of one or more messages and, as long as peer knows how to read and write **IMC** messages, it can use any of its communication links to discover and communicate to other **IMC** peers.

2.3.1 IMC Message Structures

The **IMC** protocol is not closed. New messages can be added to the protocol by changing an XML file in a central source code repository [dSeTS09a]. This happens becuase the entirety of the protocol is defined in one XML file: the various field types, header fields, footer fields and all possible message types (different body fields). The fields are specified using C types and they are also serialized similarly. However, **IMC** does not require a specific data endianess (saving some CPU cycles for embedded systems). There are some more complex types such as:

Plain Text: A character String encoded in UTF-8.
Raw Data: Raw byte array.
Message: An inline message (stripped of header and footer).
Message List: An list of inline messages.

The Table 2.3 shows the fields which are common to every message. All messages start with an header that encodes the message type, its size, generation timestamp, size, source and destination. Also, all messages start with a synchronization number which helps identifying different

Field	Type	Description
sync	uint16_t	Used for stream synchronization and determining endianess
mgid	uint16_t	Identifies the message type (payload fields)
size	uint16_t	Payload size in bytes, can be used for skipping this message
timestamp	fp64_t	Time when message was generated (UNIX epoch)
src	uint16_t	Unique identifier of the system that generated this message
src_ent	uint8_t	Identifier of the subsystem that generated this message
dst	uint16_t	Unique identifier of the system this message is addressed to
dst_ent	uint8_t	Identifier of the subsystem this message is addressed to
...		
checksum	uint16_t	CRC16 checksum of the message contents

Table 2.3: Message Structure for **IMC** Version 5.x

protocols versions that may co-exist in a single network and also allows detection of big or little-endian codification. Whenever a peer doesn't understand a synchronization number it can drop that message and wait for a message with understood protocol.

The XML specification of the **IMC** protocol is used to generate bindings for both C++ and Java. In C++ the code is generated using a series of Python scripts while for the Java version (IMCJava project) the bindings are generated using the JavaPoet project from Square. As an example, the generated Java code for a simple message with just one field can be seen in figure 2.13.

This simple serialization mechanism is used for both communication and logging. As such, a log file is a simple concatenation of all binary messages together with the XML specification of the **IMC** protocol used to produce those structures.

2.3.2 Addressing and Peer Discovery

All **IMC** messages contain source and destination fields in their headers. Both the source and destination include a top-level identifier (`src`, `dst` fields), which usually identifies a physical platform, and an application identifier (`src_ent`, `dst_ent` fields). These identifiers were kept separate from any underlying transport identifier (such as IP address or MAC) on purpose: **IMC** is not tied to any specific transport.

As such, **IMC** defines a special identifier for broadcast (`0XFFFF`) and it is up to the transport to allow broadcasting of messages for initial peer discovery. For instance, in TCP/UDP this is usually implemented via UDP multicasting but for Iridium/HTTP a central directory of peers is used.

```
1 http://10.0.10.100:8080/dune;
2 udp://10.0.10.100:6002/;
3 tcp://10.0.10.100:6002/;
4 acoustic+seatrac://lauv-xplore-2/;
5 iridium://300234061464450/;
```

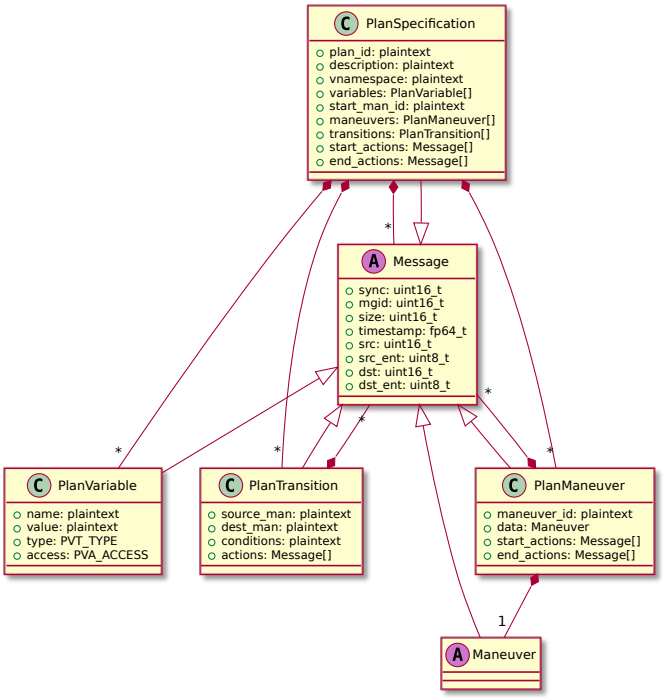

Figure 2.12: Generated PlanSpecification class diagram

```java
public class PH extends Message {

    @FieldType(type = IMCField.TYPE_FP32)
    public float value = 0f;

    public byte[] serializeFields() {
        try {
            ByteArrayOutputStream _data = new ByteArrayOutputStream();
            DataOutputStream _out = new DataOutputStream(_data);
            _out.writeFloat(value);
            return _data.toByteArray();
        } catch (IOException e) {
            return new byte[0];
        }
    }

    public void deserializeFields(ByteBuffer buf) throws IOException {
        try {
            value = buf.getFloat();
        } catch (Exception e) {
            throw new IOException(e);
        }
    }
}
```

Figure 2.13: Generated code for PH message

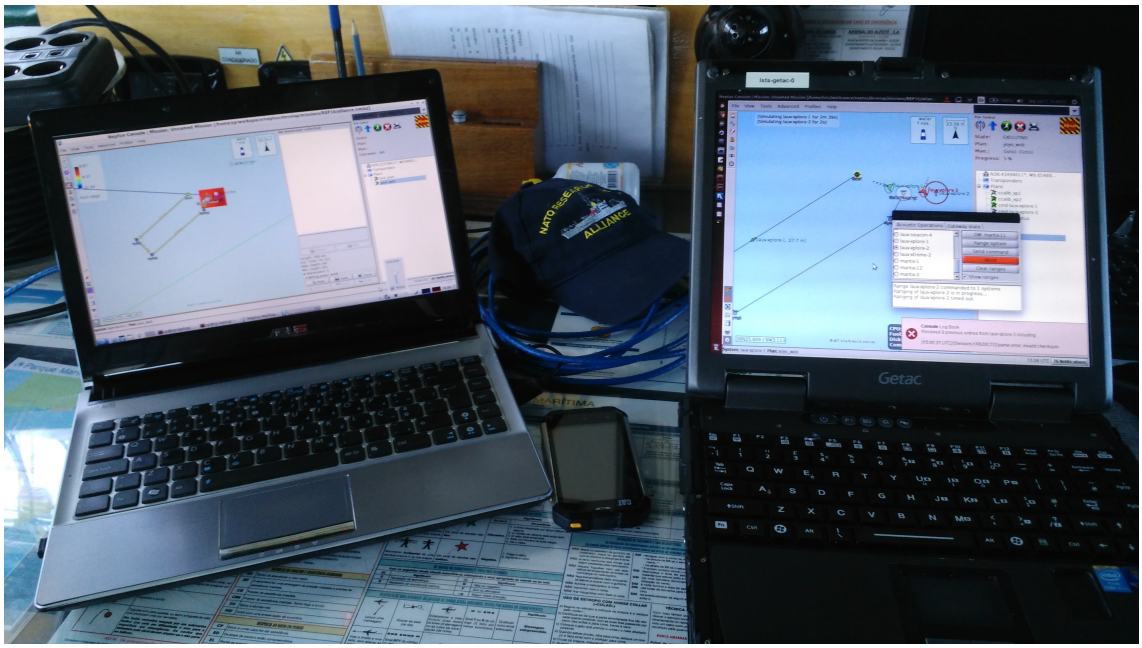

Figure 2.14: **NEPTUS** Consoles being used in the field onboard Navy ship

---

Listing 2.1: IMC transport URLs that announced by a LAUV vehicle

For discovery, **IMC** uses the `Announce` message which encodes (among other fields) the type, name and transports implemented by the nodes (together with an address specific to each transport). The supported transports are encoded as a list of URLs in the `services` field of this message. As such it is possible (and normal) to have multiple active communication means (see 2.1 for some examples) for one single peer and it is up to the protocol implementer to do proper routing.

2.4 Neptus Command and Control Infrastructure

NEPTUS is a software framework for developing user applications that interface Networked Vehicle Systems [DFG+05, PDG+06]. **NEPTUS** is developed in Java for portability and originally because of its good support for graphical applications (Java Swing). **NEPTUS** development started in 2004 when it was created to support the operation of 1 ROV and 1 AUV but it evolved into a user-friendly vehicle-agnostic tool. **NEPTUS** is currently used for operating fleets of heterogeneous autonomous vehicles not only by specialized technicians but also biologists, oceanographers, archaeologists, students and military staff. In this section we describe some of **NEPTUS** inner workings, the **NEPTUS** operator console and focus on its planning support.

Base class / Interface	Description
ConsolePanel	These are visual components that can be used as part of a console layout or as a popup panel.
ConsoleLayer	These plug-ins overlay extra information on top of the core Map component.
ConsoleInteraction	These plug-ins provide extra information on top of the Map component, and when activated can intercept all mouse and key events on the map.
MraVisualization	Abstract data visualization that is given a log file and should generate a resulting component.
ColorMapVisualization	A more specific data visualization that renders a variable as a 2D color map.
MRATimeSeriesPlot	Line chart that displays the evolution of a variable over time.
LogTableVisualization	Displays data as a table.
VTKVisualization	3D Visualizations based on the VTK[1] engine.
MRAGanttPlot	A plot that displays finite state evolution through time.
MRALogReplay	Log replay component that receives/displays events from past mission logs.
MRALogReplayLayer	Displays log replay events on top of the map.
LogExporter	Allows exporting **NEPTUS**-compatible log files to other formats.
PlanExporter	Allows exporting **NEPTUS**-compatible plan files to other formats.

Table 2.4: **NEPTUS** plug-in base classes and respective description

2.4.1 NEPTUS Plug-in Architecture

NEPTUS uses a custom plug-in architecture which supports different plug-in types: Console Widgets, Console Daemons, Interactive Map Layers, Interactive Data Visualizations, Plots, etc (see table 2.4).

NEPTUS relies on Java Annotations to discover classes that provide pluggable extensions. The annotations not only tag classes that should be inspected for their extensions but also provide meta information such as code maturity, plug-in version and author.

NEPTUS uses a modified class loader to detect and include plug-ins dynamically into the class path (loaded libraries). Then it uses the open source Reflections[2] library to detect any extension classes. The `@PluginDescription` annotation is used to tag all plug-ins which are then stored at startup in a repository where they are sorted by implemented interfaces. At run-time, any class can query the repository for plug-in implementations of any interface and can also request instantiation of any plug-in given its name or class. For instance, given the `PlanSimulationLayer` class depicted in Figure 2.16, that class will be found when looking for `ConsoleLayer` plug-ins.

[2]https://github.com/ronmamo/reflections

Annotation name	Target Elements	Usage
@PluginDescription	Classes	Plug-in class that provides one or more extensions.
@Popup	Classes	Widget plug-in that can also be displayed on its own window.
@MapLayer	Classes	Provides extra information such as priority and default visibility to map layers.
@NeptusProperty	Fields	Configurable parameters editable by the user. **NEPTUS** provides APIs for storing the properties to disk, loading and editing them.
@Subscribe	Methods	Event handler called whenever a certain type is posted. The type is derived from the first method argument.
@Periodic	Methods	Call this method periodically. The calling frequency is provided as an annotation paramter.

Table 2.5: Types of annotations used in **NEPTUS**

```
66 @PluginDescription(name = "Plan Simulation", icon = "images/planning/robot.png")
67 public class PlanSimulationLayer extends ConsoleLayer implements
   PlanSimulationListener {
68
69     @NeptusProperty(name = "Max AUV distance", description = "Warn user if AUV
   distance exceeds this value")
70     private double maxAUVDistance = 1000;
71
72     @Subscribe
73     public void on(ConsoleEventMainSystemChange evt) {
74         refreshOverlay();
75     }
76
77     @Override
78     public void initLayer() {
79         mainPlan = getConsole().getPlan();
80         refreshOverlay();
81     }
```

Figure 2.15: Example Plug-in code that showcases **NEPTUS** annotations

All the different plug-in types must implement respectively different interfaces in order to provide usable extensions. To give an example, `ConsoleLayer` plug-ins must implement the following abstract methods:

initLayer(): Do plug-in initialization. Called when console is loaded or when plug-in is added at run-time.

cleanLayer(): Do plug-in cleaning operations. Called when console is closed or when plug-in is removed at run-time.

paintLayer(): Paint this layer on the map.

2.4.2 Communications Infrastructure

Other than the previous methods, plug-in developers will potentially be interested to react to incoming messages in order to update their internal and visual states. For that they must subscribe to events of a given type using the `@Subscribe` annotation. The event types may correspond to IMC messages (generated code based on XML definitions) or **NEPTUS**-specific events such as user selections, connectivity changes, etc.

The publish/subscribe system implemented in **NEPTUS** uses the open source Google Guava[3] library, namely its Event Bus functionality. There is one singleton `EventBus` object to which all plug-ins can post events in real-time while other buses can eventually be used for log replays along side real-time monitoring.

Different **NEPTUS** components feed the real-time `EventBus` by listening to multiple network transports such as incoming UDP datagrams, TCP data or Iridium messages. As a result, whenever an IMC message is received using any of the underlying communications interfaces, the plug-ins that have subscribed handlers for that type of event will be invoked. Guava's `EventBus` uses a set of worker threads that will disseminate all incoming events by calling their handlers while providing guarantee that no object handles more than one event at a time. The end result is quite similar to what happens in the **DUNE** software for its tasks and results in completely asynchronous message passing.

NEPTUS provides a unified interface for message sending. This `NeptusMsgManager` API can be used to post messages to the local bus (internal communications), to send to a specific external system or to multicast destinations such as all consoles, all vehicles or every connected system. Moreover, the developer can select which type of transport is preferred for delivery and if others can be used alternatively (in case the preferred fails).

Since messages are sent asynchronously, a Java `Future` object is returned to the caller. This object represents a delivery promise and can (optionally) be used to check the state of the transmission and the delivery result by the sending entity.

[3] https://github.com/google/guava

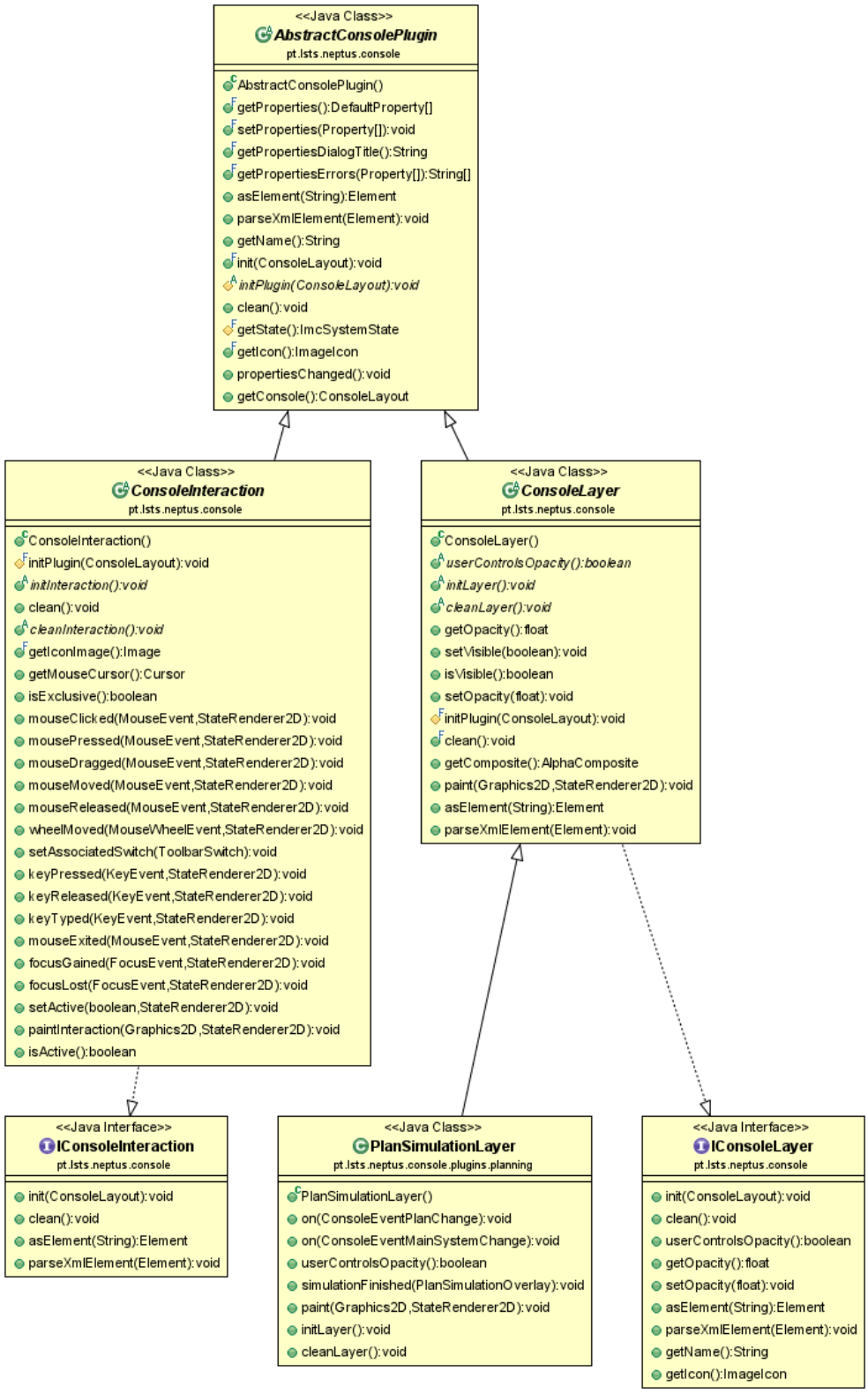

Figure 2.16: Class Diagram for (example) ConsoleLayer Plug-in

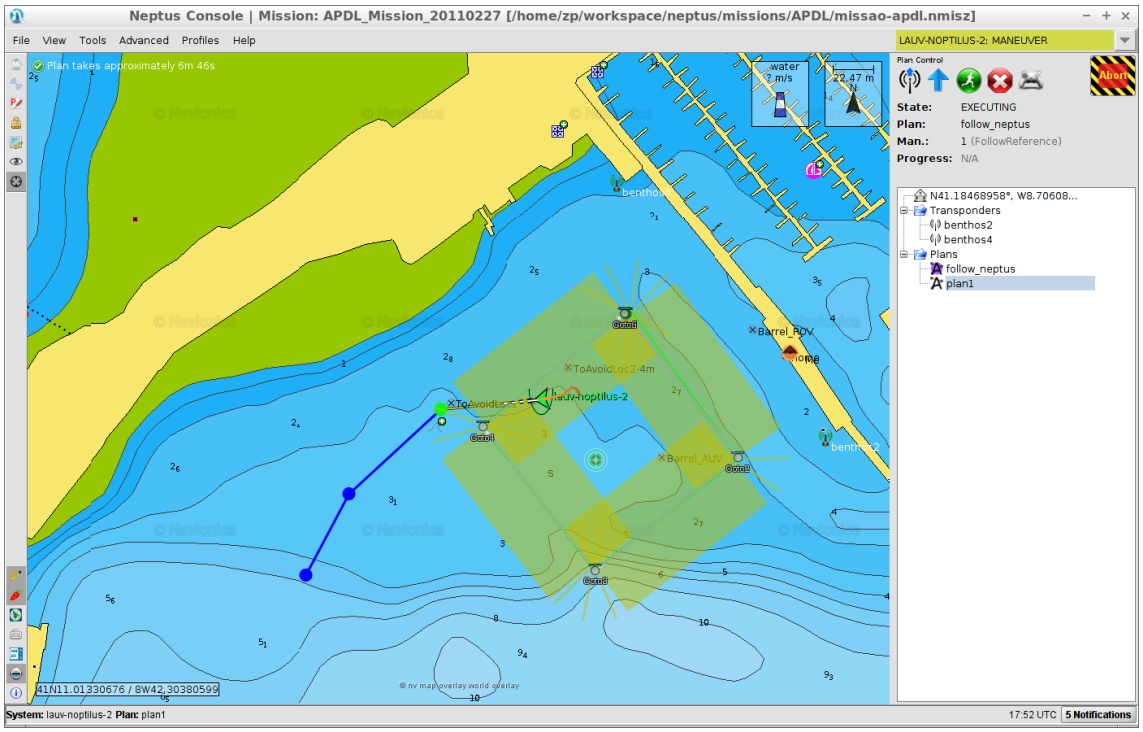

Figure 2.17: **NEPTUS** Console targetting AUV operation

2.4.3 NEPTUS Operator Consoles

NEPTUS streamlines the creation of Operator Consoles by allowing end-users to create and configure a console from scratch. When a user creates a new console, a minimal configuration with simple communications is created with no visual components, just an empty frame. It is then up to the user to add widgets such as the map component, different map layers, control buttons, etc.

This approach was used because of **NEPTUS** very heterogeneous user base: software developers, navy staff, scientists, pilots, among others. As a result, multiple console configurations exist and different users choose an interface that best matches their operational needs. For instance, UAV consoles typically do not include acoustic (underwater) communication components but include things like pre-flight checklists or wind speed gauges (see figures 2.17 and 2.18).

Both consoles depicted in 2.17 and 2.18 have similar layouts which emerged from hours of testing in the field. On the left of the console window, there is the map where vehicles, map features and other layers are shown overlayed on cartographic or rasterized maps. This mapping component has a set of buttons (toolbar on the left) that are used to switch visibility of different layers. Moreover, some of those buttons also change the way the user interacts with the map (Console Interactions).

2.4.4 Mission Planning in NEPTUS

For **NEPTUS** , a mission consists of a set of geographic features, vehicle configurations and **NEPTUS** Plans. **NEPTUS** plans are essentially sequences of maneuvers that can be executed

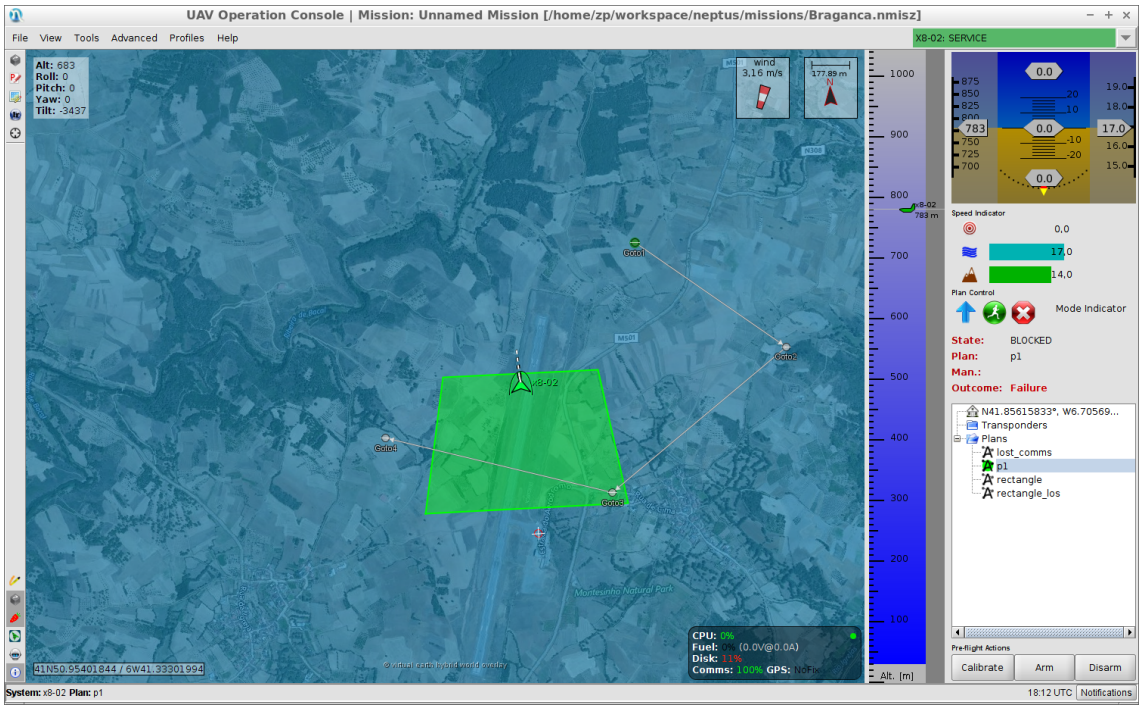

Figure 2.18: **NEPTUS** Console targetting UAV operation

by vehicles autonomously (one after another). For each maneuver, a series of initialization and termination commands can also be included such as payload reconfigurations.

NEPTUS supports the operator on the planning phase of the mission by providing user friendly plan editors where the maneuvers and their parameters can be edited visually on the map. Each maneuver has a specific set of parameters whose defaults vary from vehicle to vehicle. By looking up the vehicle configuration files, **NEPTUS** loads safe defaults for the added maneuvers.

As can be seen in table 2.6, each maneuver maps to a certain vehicle behavior and, as mentioned before, a plan is a sequential composition of the maneuvers. Without a visual planning tool like **NEPTUS**, an operator would have to be familiar not only with the implementation of every specific behavior but would also need to compose different behaviors sequentially which is a daunting and error-prone task. Instead, **NEPTUS** supports the operators by allowing plans to be edited visually according to the capabilities of the vehicles and simulated prior to execution.

On figure 2.19, the mission planning console interaction is depicted. In this mode, in which the operator interacts with the map component, it is possible to lay out the maneuvers and predict the path that should be executed by the vehicle (in yellow). The planning interaction allows the user to edit maneuvers in the map (add / remove / drag / rotate / scale) or via its parameters. In the panel to the right all parameters from the selected maneuver are shown as a list of editable properties together with a list of payload parameters that are available at the target vehicle.

Payload parameters can be set individually to each behavior in order to turn off unneeded hardware (and thus save battery) for instance during long transits between actual surveys. Moreover, this allows a single plan to have multiple objectives such as survey first with one sensor and then

Maneuver Type	Parameters	Resulting Behavior
Goto	location, z, speed	Go to *location* at specified z (depth or altitude) and *speed*.
StationKeeping	location, speed, duration	Go to *location* and stay there at surface for specified amount of *seconds*.
Loiter	location, z, radius, speed, duration	Go to *location* and stay there at specified z (depth / altitude) for specified amount of *seconds*. Moreover if the vehicle needs to move to stay at specified vertical reference, perform a circle of *radius* meters arond the *location*.
FollowPath	start location, z, additional points, speed	Follow a given path represented as *start location* and a sequence of additional *points* with geometrical offsets (x, y, z) at specified *speed*.
FollowTrajectory	start location, additional points, speed	Similar to *FollowPath*, but in this case the path points also encode desired time of arrival, which makes them a trajectory definition.
PopUp	location, duration	Go to the surface at specified *location* and wait until the navigation is corrected using GPS or *duration* seconds are ellapsed.
YoYo	location, depth1, depth2, speed	Go to *location* at specified ground *speed* while surveying the entire water column between *depth1* and *depth2* (usually performing a sawtooth or yo-yo pattern).
FollowReference	authorized source, z, speed	Start accepting reference commands from external *authorized source*. References may not specify *speed* or z in which case the provided defaults shall be used.
Rows	area, z, speed, horizontal step, ...	Survey a rectangular *area* by doing longitudinal transects (rows) spaced by *horizontal step* meters.
Elevator	radius, target z	Move vertically towards the target z value. In case of nonholonomic vehicles, it may be necessary to move in a hilicoid towards the desired z which then should have *radius* meters.
Tele-operation	authorized source	Start accepting low-level commands of thrust and heading from *authorized source*, which can be used to remotely operate AUVs for recovery or right after deployment to the water.

Table 2.6: Supported **NEPTUS** Maneuvers

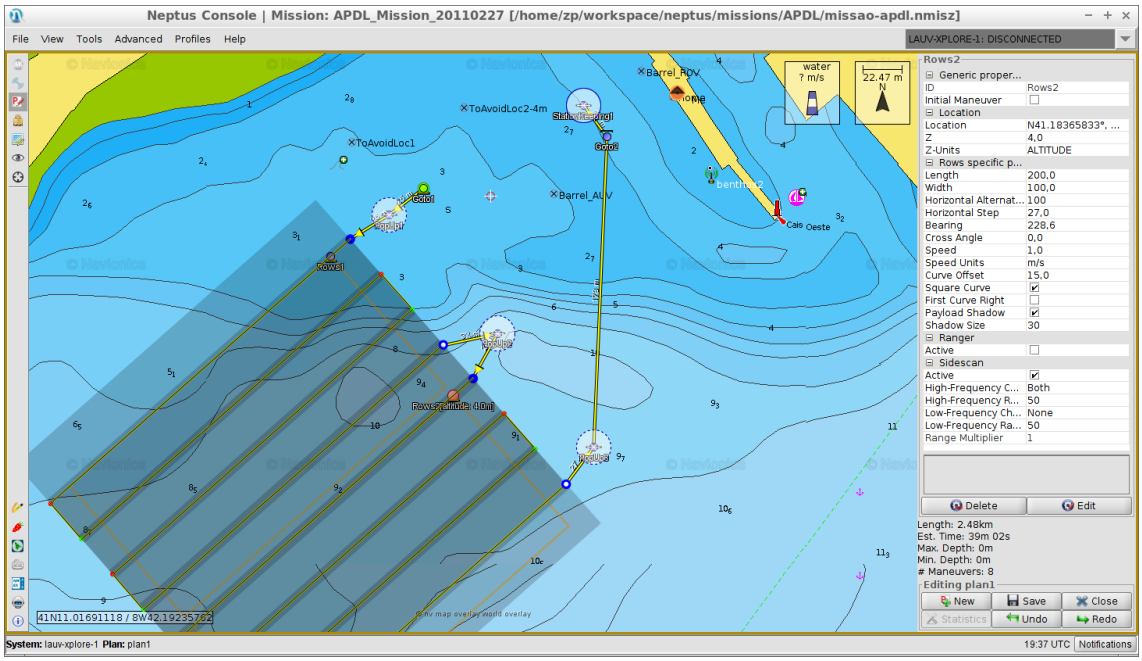

Figure 2.19: **NEPTUS** Mission Planning interface

another using different parameters.

Plan statistics such as travelled distance, maximum depth required or time required are also updated automatically whenever the plan is changed and displayed to the operator on the right. These statistics however, are based only on the path specification that results from plan maneuvers as if they were perfectly executed by an holonomic vehicle, which despite useful can sometimes lead the user into errors. A more advanced plan simulation engine (described in the next subsection) can be used instead to not only get more accurate statistics but also detect potential problems with the plan.

2.4.5 Plan Simulation and Validation Engine

As described in the previous section, despite the sequential nature of Neptus plans, its understanding can be daunting for novice and even advanced operators. Creating a plan requires not only checking that the resulting behavior is safe but also that it produces the desired results such as complete sensor coverage of an area or timely recovery. Moreover, while a plan is being executed the vehicles may stay entirely disconnected for long periods of time and it is desirable to estimate as accurately as possible what disconnected vehicles shall be doing at any time. For this reason a plan simulation engine was added to Neptus that not only allows quick overview of how a plan shall be executed by a specific vehicle but also estimate during execution what the disconnected vehicles should be doing.

The Plan Simulation Engine is based on a simple unicycle dynamics model (depicted in figure 2.20). For each vehicle a top speed as well as maximum change rates for heading (ω) and pitch are stored in a configuration file. In order to know exactly what is the desired behavior for each

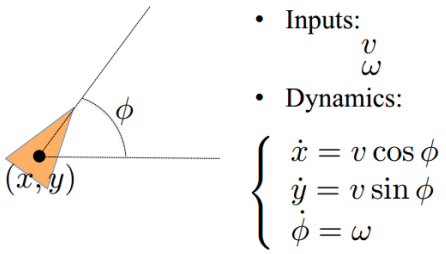

Figure 2.20: Unicycle dynamics model

maneuver, there is correspondence between maneuvers and their behavior implementations (Java classes) or simulated controller.

Simulated controllers provide an implementation for the interface `IManeuverPreview` (figure 2.21) that effectively *drives* the vehicle using the unicycle model. To simulate a plan execution, for each maneuver in the plan, the associated simulated controller is instantiated and simulated iteratively until the maneuver is done after which the subsequent maneuver (and respective controller) is instantiated and plan simulation is resumed. Moreover, for each simulation step, the controller can generate an execution state which can be used to register what has been done for that maneuver at that state. This execution state will be passed back to the controlled if the execution is to be resumed from that point. This follows the *Memento* behavioral design pattern [Gam95]. This is especially important when the simulation engine is used together with actual vehicle execution as we describe next.

```
public interface IManeuverPreview<T extends Maneuver> {

    public boolean init(String vid, T man, VState state, Object manState);
    public VState step(VState state, double timestep);
    public void reset(VState state);
    public boolean isFinished();
    public Object getState();
}
```

Figure 2.21: Neptus `IManeuverPreview` interface

2.4.5.1 Estimating vehicle state during execution

Whenever **NEPTUS** detects that a vehicle is executing a plan autonomously, it creates a simulator to track what the vehicle should be doing. This is mostly important when the vehicle disconnects from the base station in which case the simulator is used by the operator to predict its present state.

Whenever a new state from the vehicle is received by **NEPTUS** , the simulation engine is updated and must synchronize its internal (simulation) state. The approach to determine the maneuver currently being executed is rather simple: from the list of all states generated in the simulation select the one which is closer to the received state. The distance function takes into account

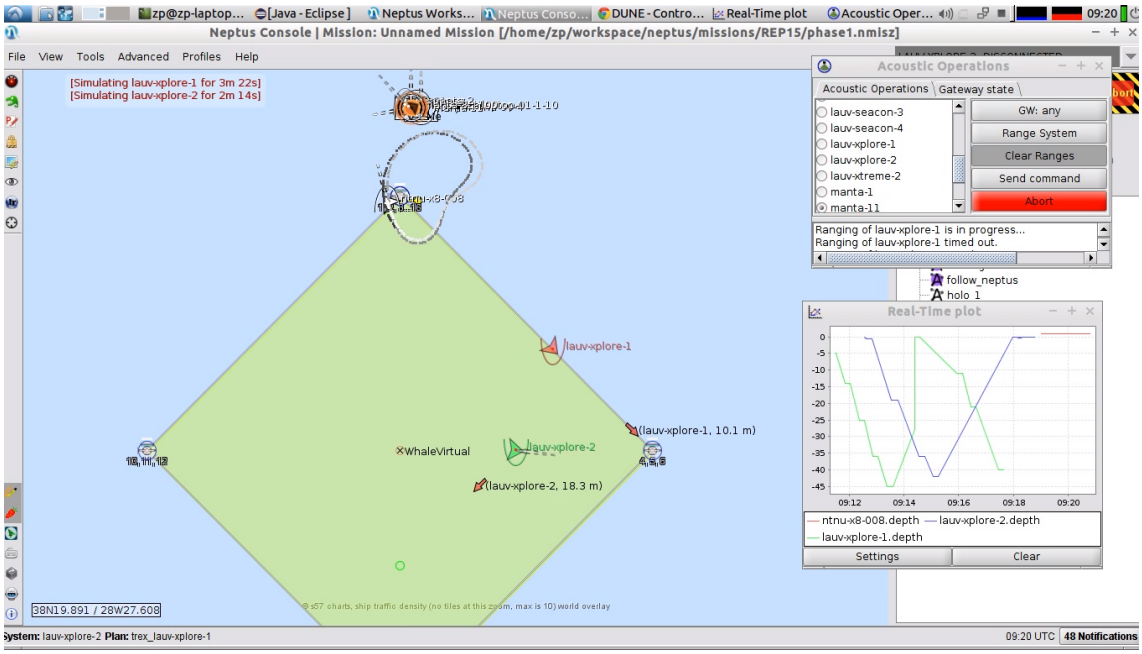

Figure 2.22: Real-world mission execution with only simulated states for AUVs lauv-xplore-1 and lauv-xplore-2

not only euclidian distance but also the difference in the heading of the two states. Then, after selecting the closest state which will identify the simulated controller, its memento together with the received position is used to restart the simulator.

The result of this simulation can be effectively seen in figure 2.22 where two AUVs are executing an autonomous mission and thus only sporadically a state is received via acoustic modem. As a result two simulator instances are being used to predict the state of each vehicle underwater: displayed on the map is the position and predicted depths of the two vehicles.

Chapter 3

Approach

In this chapter we describe 3 different approaches that were tested for mixed-initiative planning of autonomous vehicles. The first approach, described in section 3.1, consists in using a centralized domain-independent planning engine to create a multi-vehicle plan according to perceived capabilities and high-level tasks provided by the human operator. The second approach described in section 3.2 uses onboard plan deliberation to generate and adapt plans inside the autonomous vehicles in response to user submitted objectives or potential failures. Finally, section 3.3 merges the previous two approaches by having onboard plan deliberation and adaptation as well as a centralized planner that generates high-level objectives for a network of heterogeneous vehicles.

3.1 Centralized Planning in Neptus

In this section we describe a first approach towards a mixed-initiative planning and control tool-set that combines "high-level" goal-oriented mission planning and "low-level" control of the vehicles. Neptus is continuously receiving telemetry and other data from all deployed vehicles. As such, it can construct a knowledge base of capabilities that exist in the network and provide this information to AI planners that, given a set of user-defined objectives, can generate the actions (maneuvers) to be executed by the individual vehicles.

3.1.1 Plan Representation

We use PDDL 2.1 [FL03] for representing our planning problems. The environment is described by predicates and numeric functions. Actions are specified via preconditions, effects and the duration of their execution. Preconditions are sets of logical expressions that must be true in order to have the actions executable. In PDDL, these expressions can take place prior to starting action execution, prior to finishing action execution, or over the whole time period when the action is executed. Effects are sets of literals and function assignments that become true when the action is executed. In PDDL, effects can occur just after starting, or just after finishing action execution.

The PDDL representation allows us to specify domain models and problem specifications separately. Usually, one domain model is used for a class of problems. In particular, a domain model

consists of predicate and function descriptions and action definitions, while a problem specification consists of definition of objects, an initial state and a set of goal conditions.

In domain-independent planning, planning engines and domain models are decoupled. A planning engine can deal with various domain models, and a domain model can be accepted by various planning engines. Therefore, if the domain model is modified, there is no reason to modify the planning engine; equally it is possible to replace one planning engine with another without changing the domain model.

Specifying problems in `PDDL` is relatively straightforward. For example, predicates are defined in the form of `(at noptilus1 task1-loc)` and function assignments are defined in form of `(= (battery-use noptilus1) 50)`. A plan action is in the form `48.0: (MOVE XPLORE1 XPLORE1-DEPOT T01-LOC) [1365.000]`, where the first number refers to the time-stamp when the action is executed, while the second refers to the duration of action execution. Every action accepts a single vehicle as a parameter.

In this work we use **LPG-TD** [GSS03] as the planning engine. **LPG-TD** is based on a local search in Planning Graphs [M. 04] which allows executing actions that do not interfere with each other in each plan step. While it is a mature planner, most state-of-the-art planning engines either do not support required features (such as numeric functions or durative actions) nor do they scale well. For instance, while both **Optic** [BCC12] and **EUROPA**$_2$ [FJ03] offer richer representations, our experimentation with these demonstrated that their performance did not scale well for problems we envision for such mixed-initiative interaction.

3.1.2 Domain Model for Networked Vehicle Systems

We have conceptualized mission requirements in the form of a domain model specification. This conceptualization is divided into three categories: object types, predicates and functions, and actions similar to work of Shah et al. [SCK+13]. Object types refer to classes of objects that are relevant for the planning process such as: *vehicle* (V), *payload* (P), *phenomenon* (X), *task* (T), *location* (L). By "phenomenon" we mean a target object or area of interest, where we assume that such phenomena are deterministic and static. Tasks are considered as atomic, to be fulfilled by a single vehicle.

Predicates and functions describe states of the environment. In particular, predicates represent relationships between objects, and functions refer to quantity of resources related to the objects. In our case, we have defined: $at \subseteq V \times L$ – a location of the vehicle, $base \subseteq V \times L$ – a location of the vehicle's *depot*, i.e. known positions where a vehicle is expected to be safe when on the surface, $has \subseteq V \times P$ – whether a payload is attached to the vehicle, $at\text{-}phen \subseteq X \times L$ – a location of the phenomenon, $task \subseteq T \times X \times P$ – which describes a task of getting data about a phenomenon from a specific payload, $sampled \subseteq T \times V$ – whether data of a given task has been acquired by the vehicle, $data \subseteq T$ – whether the task data has been acquired from the vehicle, $dist : L \times L \rightarrow R^+$ – a distance between two locations, $speed : V \rightarrow R^+$ – speed over ground of the vehicle, $battery\text{-}level : V \rightarrow R_0^+$ – the amount of energy in a vehicle's battery, $battery\text{-}use : V \cup P \rightarrow R^+$ – battery consumption

per distance unit (moving a vehicle) or per time unit (using a payload). We currently make an assumption of linear energy use both for moving or using a payload.

Actions when executed modify the environment according to their effects. We have specified 4 actions (we denote t_s as time when an action is executed, and t_e as time when an action execution ends):

- *move*(v,l_1,l_2) – the vehicle v moves from its location of origin l_1 to its desination location l_2. As a precondition is must hold that in t_s: $(v,l_1) \in at$, *battery-level*$(v) \geq dist(l_1,l_2) *$ *battery-use*(v); and in t_e: $\neg\exists v_x \neq v : (v_x,l_2) \in at$. The effect is that in t_s: $(v,l_1) \notin at$, and *battery-level*$(v) =$ *battery-level*$(v) - dist(l_1,l_2) *$ *battery-use*(v) , and in t_e: $(v,l_2) \in at$

- *sample*(v,t,x,p,l) – the vehicle v samples a phenomenon x by the payload p. As a precondition it must hold that in t_s: *battery-level*$(v) \geq (t_e - t_s) *$ *battery-use*(p), and in $[t_s,t_e]$: $(v,l) \in$ *at*, $(x,l) \in$ *at-phen*, $(v,p) \in$ *has* and $(t,x,v) \in task$. The effect is that in t_s: *battery-level*$(v) =$ *battery-level*$(v) - (t_e - t_s) *$ *battery-use*(p), and in t_e: $(t,v) \in sampled$.

- *survey*(v,t,x,p,l_1,l_2) – the vehicle v surveys the area (between l_1 and l_2) of a phenomenon x occurrence by the payload p. As a precondition is must hold that in t_s: $(v,l_1) \in at$, *battery-level*$(v) \geq dist(l_1,l_2) *$ *battery-use*$(v) + (t_e - t_s) *$ *battery-use*(p), and in $[t_s,t_e]$: $(x,l_1) \in$ *at-phen*, $(x,l_2) \in$ *at-phen*, $(v,p) \in$ *has* and $(t,x,v) \in task$. Also, no other vehicle can perform the survey action over the phenomenon x in $[t_s,t_e]$. The effect is that in t_s: $(v,l_1) \notin at$, and *battery-level*$(v) =$ *battery-level*$(v) - (dist(l_1,l_2) *$ *battery-use*$(v) + (t_e - t_s) *$ *battery-use*$(p))$, and in t_e: $(v,l_2) \in at$, $(t,v) \in sampled$.

- *collect-data*(v,t,l) – the data associated with a task t is collected by vehicle v. As a precondition it must hold that in $[t_s,t_e]$: $(v,l) \in at$, $(v,l) \in base$ and $(t,v) \in sampled$. The effect is that in t_e: $t \in data$.

As an example, the `Sample` action is encoded as depicted in Fig. 3.1.

```
(:durative-action sample
 :parameters (?v - vehicle ?l - location
    ?t -task ?o - phenomenon ?p - payload)
 :duration (= ?duration 60)
 :condition (and (over all (at-phen ?o ?l))
   (over all (task ?t ?o ?p))
   (over all (at ?v ?l))
   (over all (having ?p ?v))
   (at start (>= (battery-level ?v)
             (* (battery-use ?p) 60))))
 :effect (and (at end (sampled ?t ?v))
   (at start (decrease (battery-level ?v)
             (* (battery-use ?p) 60))) )
```

Figure 3.1: The `Sample` action in `PDDL`.

The user specifies mission tasks in **NEPTUS**'s front-end by pointing to the locations/area of the phenomena the user wants to observe and by selecting payloads the user wants to use for obtaining data. AUVs that are connected to **NEPTUS** are considered as operational and could be used for a mission. **NEPTUS** then encodes the information about vehicles and tasks into `PDDL`, and as a goal sets to acquire all the data specified by the tasks.

The planner decides which vehicle does which task and in which order the tasks are to be performed. Plans follow constraints specified in the action descriptions, i.e., collision avoidance (at most one vehicle can be in one location) and energy constraints (vehicles must not run out of energy before finishing all the tasks) and plans are optimized for total mission time. If the planner is unable to find a plan, the user is notified of plan failure, requiring her/him to iteratively relax constraints. An illustrative example follows:

Example: An operator needs to plan two AUVs (#'s 1 & 2). #1 has multi-beam, #2 has the downward looking camera, and each of the vehicles has a side-scan sonar. Then, the operator specifies three tasks; two scope out areas of interest, where one has to be surveyed by side-scan and the second by multibeam, while the third refers to a location of another phenomenon that has to be sampled by camera. The planner then assigns the multibeam related task to be performed by #1 and the camera task to be performed by #2. The task where side-scan sonar is required might be performed by both AUVs. The planner decides to allocate it to #2 since it takes less time than #1. However at plan time, it is determined that #2 does not have enough energy; the task therefore gets assigned to #1. When neither of the AUVs has enough energy, the planner does not return any plan, since this last task cannot be allocated.

3.1.3 Integration with Neptus

As shown in Fig. 3.2, a user specifies the mission in **NEPTUS**, which then automatically generates planning problem description that is accepted by the planning engine. Given the domain model and problem specification, the planning engine returns a plan (if one exists) to **NEPTUS** which in turn, distributes the plan among the vehicles, where it is executed. Since the planning engine is used as a "black-box" we have extended **NEPTUS** in order to generate problem specification in `PDDL` as well as to process `PDDL`-compliant plans (see Figure 3.2).

The **NEPTUS** integration was done via a newly developed `ConsoleInteraction` plug-in whose class diagram can be seen in Figure 3.5.

In order to produce the initial state from **NEPTUS**, this plug-in uses a **NEPTUS** API to query which payloads (capabilities) are available for all connected vehicles.

The same **NEPTUS** plug-in is used to to place high-level tasks in the map visually. The human operators can either place locations to be visited with a certain sensor or rectangular areas to be surveyed with one or more sensors. Instead of allowing the user to define the low-level payload settings such as side-scan sonar frequency or multi-beam sonar range, this interface allows only to select which type of sensor to use and **NEPTUS** selects the best parameters for the payload instead.

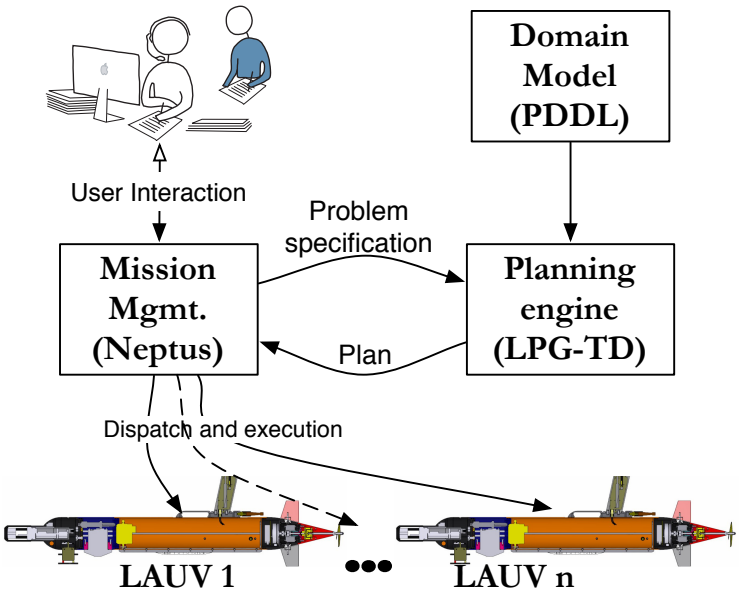

Figure 3.2: A modular architecture of the system.

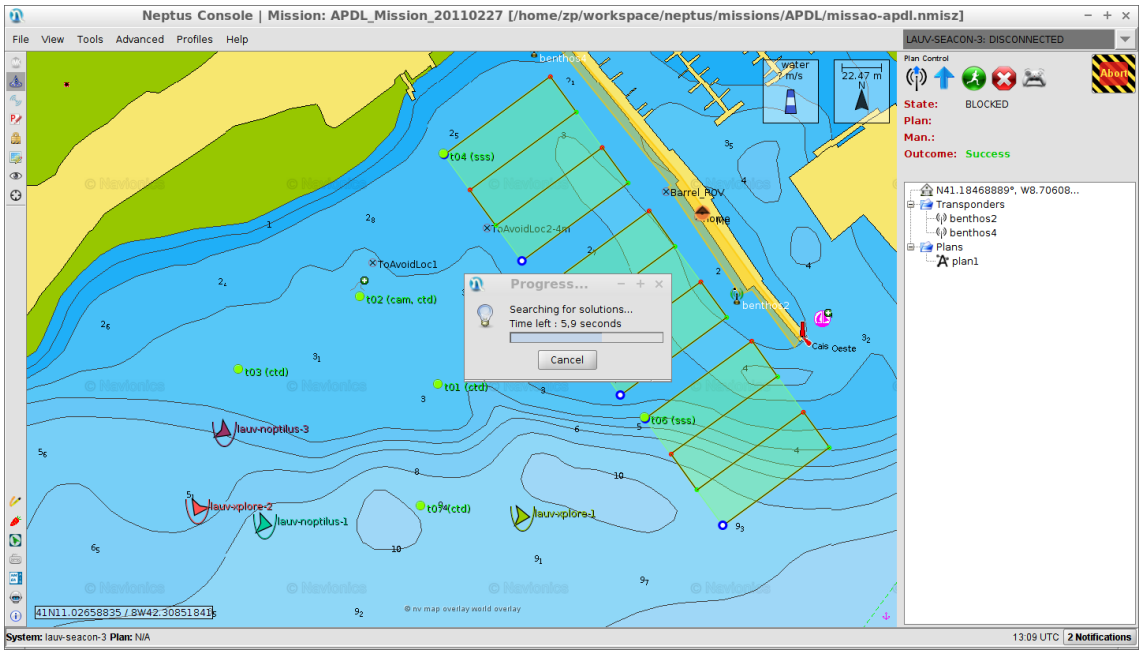

Figure 3.3: Snapshot of the centralized planner plug-in being used to generate plans for multiple vehicles.

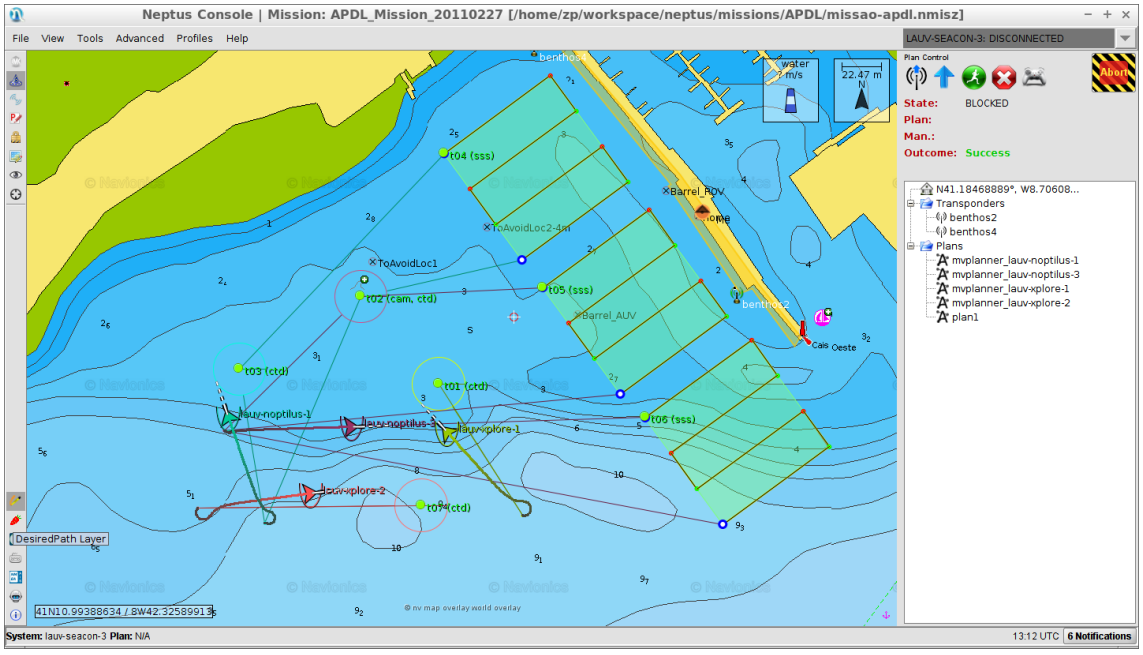

Figure 3.4: Snapshot of vehicle execution of the resulting generated plans.

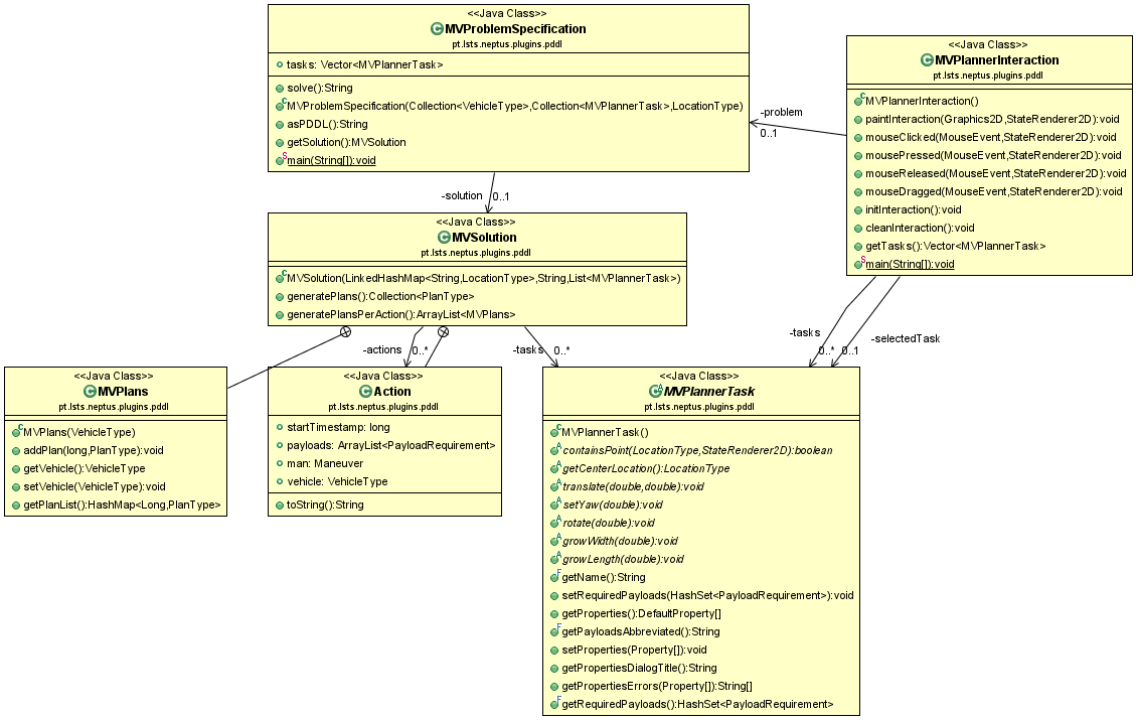

Figure 3.5: Problem Specification Implementation Class Diagram.

After any number of tasks have been added to the map, the user can invoke the **LPG-TD** planner by selecting the "generate plans" option. As a result, an object of type `MVProblemSpecification` is created / updated. The resulting object includes information about connected vehicle capabilites as well as current vehicle positions and the tasks provided by the user (see Figure 3.5). This class provides methods to translate this information to `PDDL` so that it can be used as the planner initial state.

The problem file (encoded as `PDDL`) requires informations such as the locations of all tasks and vehicles, execution duration of the tasks as well as time required to move between every two locations. For calculating the times we use the **NEPTUS** simulation engine described in section 2.4.5.

The `MVProblemSpecification` also includes the method `getSolution()` which is used to invoke an external **LPG-TD** solver with the generated initial state and obtain the best solution. The solution score is currently based only on the time: solutions that require less time are considered better. The resulting solution is encoded as a `MVSolution` object which can be used to generate **NEPTUS** plans to be executed by the connected vehicles.

Plan generation translates the various found `PDDL` actions into different maneuvers. For instance, *move* actions are translated to `Goto` maneuvers, *survey* actions are translated into `Rows` maneuvers, *sample* actions are translated into `Loiter` maneuvers and *communicate* actions are translated into `StationKeeping` maneuvers.

The resulting plans are added to console's mission and their execution can be ordered automatically in a second step. This two-step planning process is used to allow the operator to see the generated plans before execution and get his/her aproval before automatic execution. This was added in order to allow the operator to change any task requirements if the found solution is not desirable for any reason.

After the plans are commanded to the vehicles they are executed in its full extent without interaction with **NEPTUS** or the operator. As a result, potential delays or errors while executing one of the tasks are propagated to the entire plan execution. That is because there is no onboard plan adaptation. In the following subsection we describe a new approach that uses onboard planning to generate and adapt the behavior of the vehicle onboard.

3.2 Onbard Planning and Execution

Instead of creating or generating the plans beforehand, researchers at the Monterey Bay Aquarium Research Institute (MBARI) have designed, tested and fielded the Teleo-Reactive EXecutive (**T-REX**) [MPR+08, PRM10, RPB12], a plan execution framework that uses temporal Constraint-based Planning [M. 04] with plans synthesized onboard the vehicles, both before and while executing the mission autonomously and according to a set of high-level objectives. The generated plans are, by definition, a valid sequence of actions according to a given domain model, which enforces the safety of the resulting behavior.

Leveraging that work, we have extended the LSTS toolchain in order to encompass an onboard planner/executive based on MBARI's **T-REX** and NASA's **EUROPA** planner. **T-REX** is a mission executive that allows different reactors (threads of execution) to interact with other reactors by providing internal (controlled) timelines and using external timelines, controlled by other reactors'. At the core of the **T-REX** framework is the deliberation engine, **EUROPA**, which is a versatile constraint-based temporal planner which continues to be deployed on a diverse set of applications including recently, planning for the International Space Station.

EUROPA is an AI planner originally developed NASA AMES as an evolution of the planner that was used to generate plans to be executed onboard the Mars rovers on the MAPGEN system. **EUROPA** is a vastly re-factored, higher performance open-source version of it.

EUROPA uses a domain model written in the `NDDL` (Novel Domain Description Language), together with initial conditions and goals (also in `NDDL`), to construct a set of *temporal relations* that must be true at start time. These models include assertions about the physics of the vehicle, i.e., how it responds to external stimulus and internally driven goals.

By propagating the temporal relations forward and applying goal constraints, **EUROPA** can select a set of conditions that should be true in the future, where some of these conditions will correspond to actions the agent must take. The planner can backtrack and try another path during the search process if a goal cannot be reached while being capable of discarding unreachable goals [Set12].

3.2.1 Temporal Planning Essentials

A temporal plan is composed of a temporally scoped predicate called a *token*. A token can be defined as a property – described as a first order logic predicate – with its associated temporal scope $(start, duration, end)$ using flexible interval arithmetic. All the attributes of a token are described as a domain of possible values for this token in the plan context. In order to be part of the plan, a token needs to be associated with one of the plan timelines.

A *timeline* is a sequence of tokens describing the evolution of a state variable. Concurrency between timelines and therefore between tokens on separate timelines, is the basis for concurrent state variable evolution. Tokens are causally linked by rules in a *domain model* that describe temporal relations and/or causality links between tokens [FJ03, PRM10]. Finally a token can be marked optionally as either being a *Fact* or *Goal*; while a Fact requires no justification a Goal will need not only to be inserted in the plan but necessarily have a causal chain connecting it to one or more Facts.

The planner works by continuously repairing *flaws* in a plan until no more flaws are present. Prior to execution, all partial plans must not have any flaws. Typically we deal with two types of flaws[1]:

[1]There can be additional flaw types depending on the solver being used; but these are the minimum needed to converge to a solution.

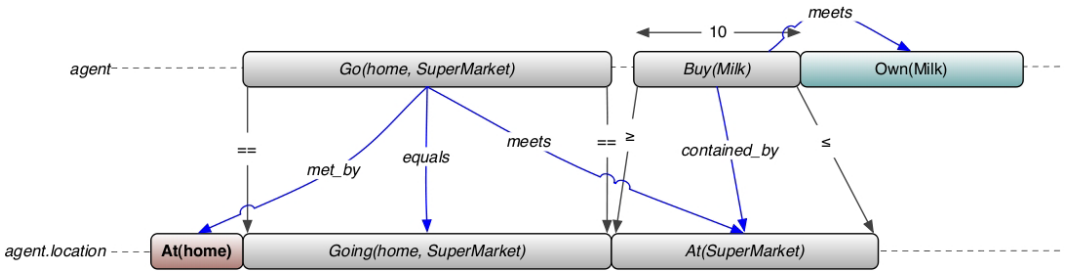

Figure 3.6: Timelines and possible relationships.

- **Open condition:** A token is not yet associated with a timeline. It can be resolved by either inserting the token into a specific timeline, or merging with a compatible token
- **Threat:** Once a token has been inserted it may impact other tokens indirectly through possible overlapping requirements. A timeline by definition, enforces a strict sequence of tokens with no concurrency within the timeline. The solver then needs to enforce a scheduling constraint on those potentially conflicting tokens so they cannot overlap, for example, by enforcing that one should occur before the other.

The solver resolves these flaws until either it reaches an inconsistency (i.e. a situation where constraints of the plan cannot be satisfied), in which case the planner will backtrack to explore alternate solutions; or a consistent solution is found and the plan presents no further flaws generating a valid solution.

Note that while the plan might be complete, it does not have to commit to the value of its variables. For example, the start time of a token can be left to be the interval $[1, 10]$ as long as it does not present a threat to the partial-plan. This leaves the decision of the start time to the executive which is critical for operating in uncertain real-world environments.

3.2.2 Deliberative Planning in T-REX

For **T-REX** , a reactor is an interacting component that has an internal state that is dependent on external observations provided by other reactors. Such information exchange occurs by sharing state variables over a common temporal horizon. A state variable describes the evolution of an attribute of the agent over time. The concept of reactor state evolution (timelines) used by **T-REX** is similar to that of **EUROPA**'s temporal relations which allows for very good integration between **T-REX** and **EUROPA**.

T-REX timelines hold tokens that have a type, starting time and duration (possibly indefinite) together with other attributes. One conceptual view of these timelines is depicted in 3.6. In it, we see two different timelines *agent* and *agent.location* that can hold different state predicates (called tokens). Reactors interact by adding observation tokens into their controlled timelines and posting goals to external timelines. Goals are similar to observations but their start time is defined in the future. If they eventually get done, they shall be posted as an observation in the timeline in which they were requested.

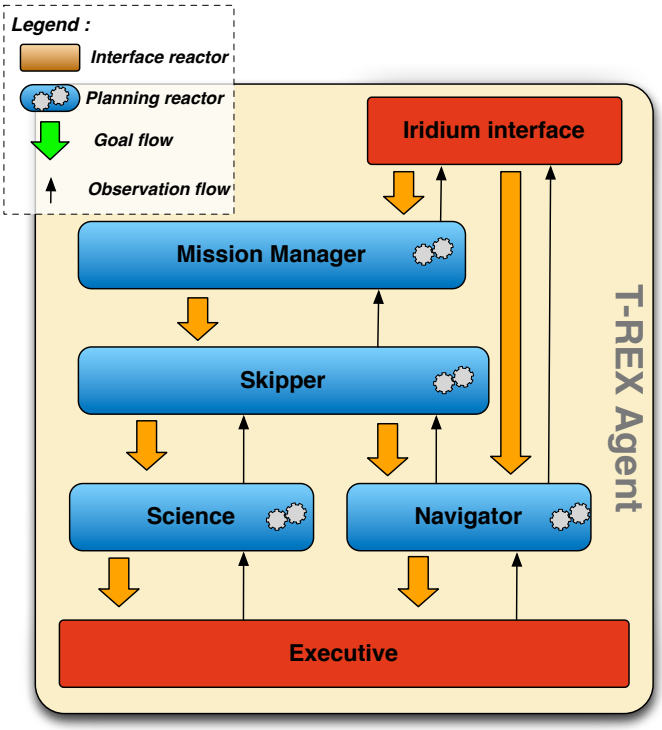

Figure 3.7: Conceptual view of an example T-REX agent composition.

Fig. 3.7 shows an instance of interacting reactors within one **T-REX** agent (running instance composed by multiple reactor components). Some of the reactors are deliberative while others are purely reactive or “interface" reactors.

3.2.3 Integration of T-REX into LSTS Toolchain

Integration of deliberative planning into the LSTS toolchain was undertaken by having **T-REX** running as a separate process inside the vehicles. This **T-REX** instance contains a set of deliberative reactors, based on the **EUROPA** planner and a platform-specific reactor. The deliberative reactors use a domain model that allows them to interface with the *Platform* specific reactor. This reactor connects to **DUNE** through **IMC** and translates incoming messages into **T-REX** observations which are posted to an internal timeline. The *Platform* reactor also accepts **T-REX** goals which are eventually translated into **DUNE** commands (sent through **IMC**). A conceptual view of this integration is depicted in 3.8.

On the **DUNE** side, there is a **T-REX**-specific task that monitors **T-REX** execution and can be activated to start accepting its commands or deactivated if onboard planning is to be paused for some reason. Moreover, whenever the user requests a plan execution on a vehicle currently being controlled by **T-REX** , that command takes precedence and **T-REX** is automatically deactivated. While **T-REX** is deactivated, it still does state synchronization but the domain model enforces that no commands can be sent to the vehicle while on the deactivated state. This allows deliberative

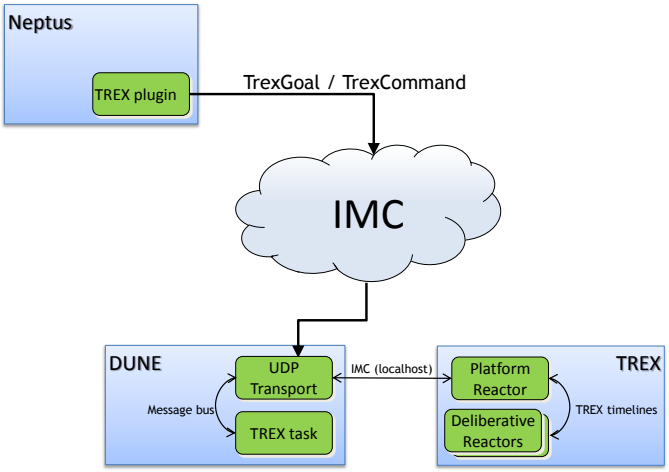

Figure 3.8: Integration of Deliberative Planning on the LSTS Toolchain.

reactors to receive any new goals and plan future actions to take when **T-REX** is reactivated on user request.

The **T-REX** implementation involved the creation of different reactors for encapsulating functionalities and thus improve flexibility. This flexibility comes from the possibility of swapping reactors with others with similar interfaces in terms of their used timelines. Next, we describe briefly the reactors that were developed for LSTS domain.

Platform: This reactor receives **IMC** messages from **DUNE** and generates corresponding observations in its controlled timelines. Moreover, this reactor also accepts maneuver goals which eventually result in the request and execution of plans in the vehicle.

Safety Bug: This simple reactor listens to all timelines and if it detects that the owner has terminated, sends an "Abort" command to **DUNE** which makes the platform interrupt execution immediately.

Navigator: This is a deliberative reactor that accepts "At" goals. As a result it will post maneuver goals that will drive the vehicle towards the desired locations.

Surface: This is a deliberative reactor that drives the vehicle towards the surface periodically by posting objectives to the Navigator reactor.

Yoyo: This is a deliberative reactor that drives the vehicle from point A to point B while measuring the entire water column. In also decides when to start ascending either to improve navigation or because it is too close to the endpoint to dive deeper.

3.2.3.1 FollowReference Maneuver

When the integration of **T-REX** and **DUNE** started, we needed to select a control interface that would allow the *Platform* reactor to command the actions of the vehicle safely. There was the option of issuing multiple maneuvers but that was very limitative as the maneuver parameters

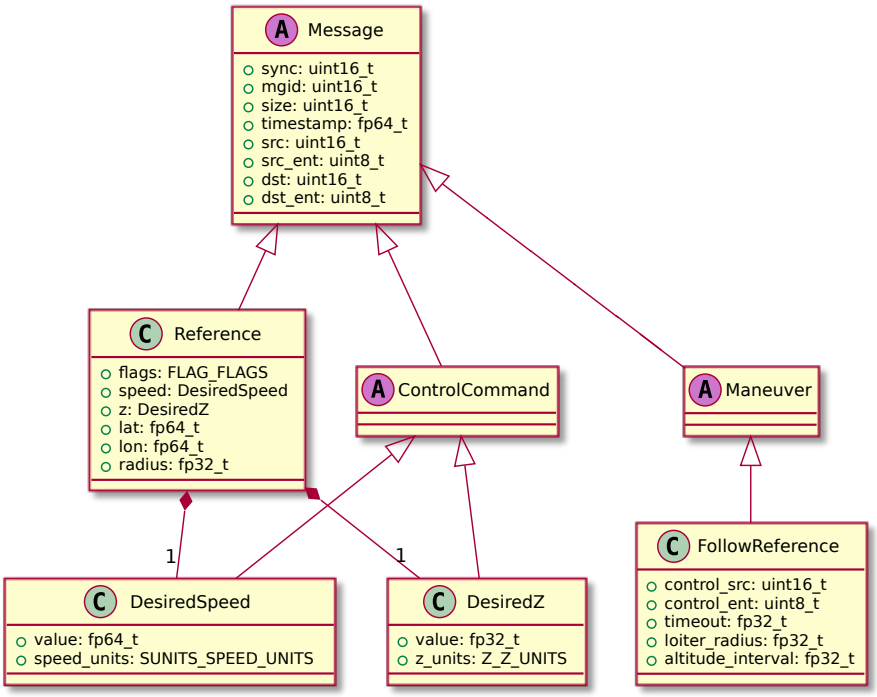

Figure 3.9: FollowReference and asspciated messages' structures.

would be constantly changing as due to **T-REX** being continuously adapting to the environment and received goals.

As a result, we devised a new maneuver that would continually accept external references and drive the vehicle towards attaining those references which we called the *FollowReference* maneuver. This maneuver accepts as parameters some default values such as default speed or default depth to use in case the external controller does not need a specific speed and also the authorized source for references.

After the maneuver is started on the vehicle, only the authorized external controller can drive its movement via *Reference* messages which encode desired target destination, loitering radius and z reference as can be seen from the class diagram extracted from the **IMC** message structures depicted in Figure 3.9.

As oposed to what happens with, for instance, the *Goto* and *Rows* maneuver implementations in **DUNE**, the implementation of *FollowReference* differs in that it verifies that both the end point as well as desired depth or altitude are attained before considering the target reference "done". It is then up to the external controller (in this case **T-REX**) to send a new reference or keep the last one but a continuous feedback loop of references and their attainability is maintained between the external controller and **DUNE**.

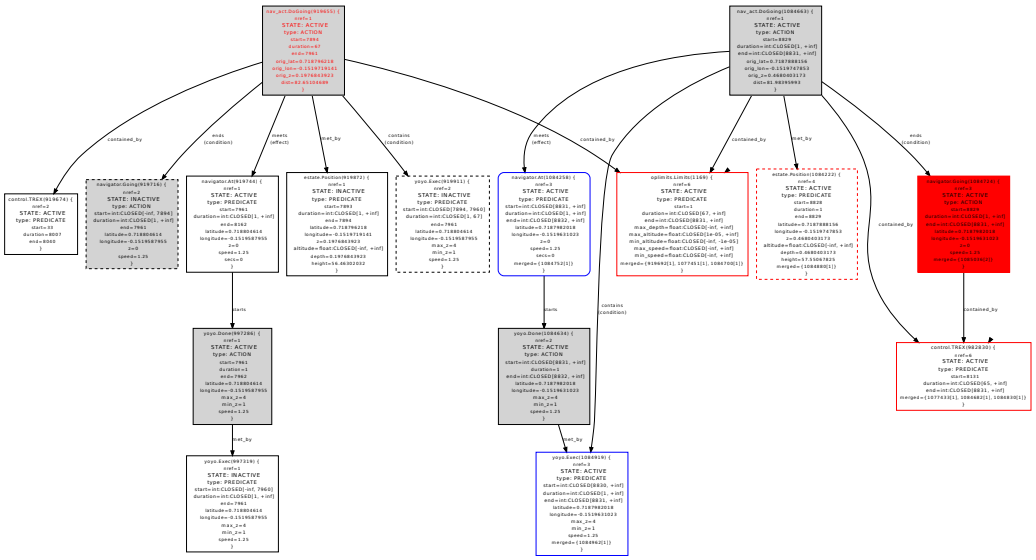

Figure 3.10: Plan instance created by **EUROPA** planner using LSTS domain model.

3.2.3.2 LSTS Domain Model

The LSTS Domain model was based upon an already existing domain model used for AUVs at MBARI. This domain model would accept as goals points to be visited inside some time window as well water column surveys around a moving point.

We adapted the existing domain model with the dynamics of the LAUV vehicles used at LSTS as well as its operational constraints such as maximum depths and periodicity between resurfaces. The domain model was also adapted in order to pause behavior generation whenever **T-REX** is deactivated either because of an hardware fault or because the user has requested so.

A glimpse over the types of plans that are generated by the **EUROPA** planner (which are continuously changing in response of received observations of the platform) can be seen in Figure 3.10. In the figure, the red tokens are active which means they are present observations and other are in the past or correspond to future goals. All the relationships between the tokens, as they are encoded in the domain model, can also be seen in the same figure.

3.2.3.3 Integration with Neptus

In order to send surveys to vehicles running **T-REX**, a new **NEPTUS** plug-in was developed which allows adding and removing goals visually. Each goal type was associated with one implementing class in **NEPTUS**, similar to what happens with different **DUNE** maneuvers.

All goal implementations extend the abstract class `TrexGoal` which defines what are the goal attributes (set by the user) and translation to and from **IMC**. Moreover, this class also allows translating the goal into a **NEPTUS** plan that allows a rough preview of what the vehicle would do in order to execute this goal. This is however not used for simulation but just to display request goals on the map as can be seen in figure 3.11.

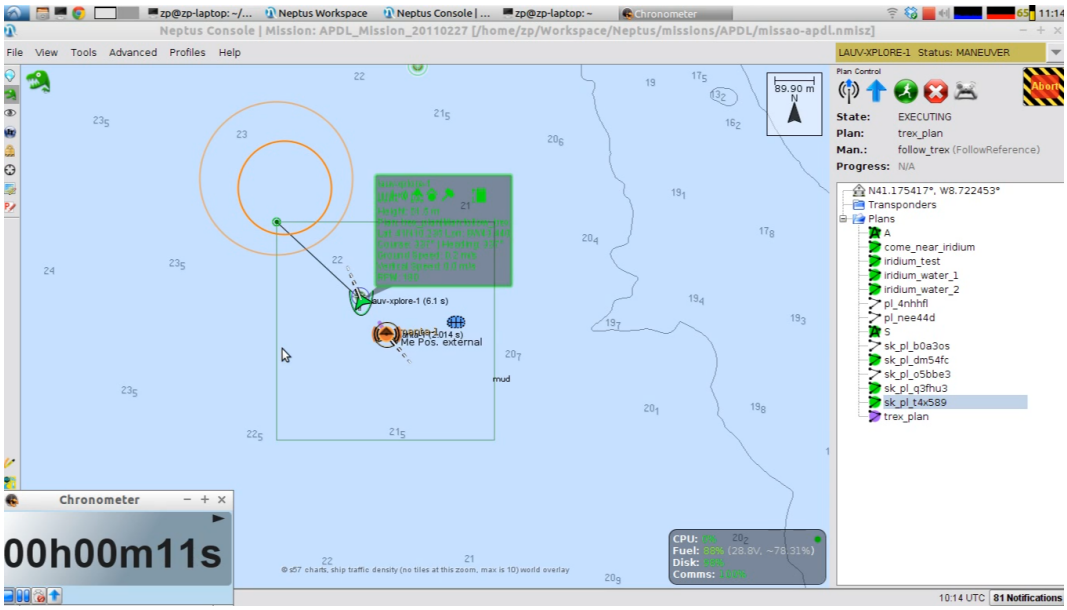

Figure 3.11: **NEPTUS** snapshot while controlling a **T-REX**-controlled AUV.

3.3 Distributed Planning

In section 3.1 we've shown how integrating **NEPTUS** with a centralized planner can be used to generate plan scripts for multiple vehicles which can execute them (blindly) from start to finish if no failures occur. Moreover, we've shown in the previous section 3.2, how **T-REX** can be used to generate plans onboard and adapt to the surrounding environment according to a planning domain model and objectives provided by the human operators. In this section we describe an integration of the two approaches where a centralized planner generates high-level objectives based on the perceived state of the world and forwards these objectives to vehicles that adapt their plans accordingly.

After developing the centralized planning and onboard planning approaches described earlier, we have designed **EUROPTUS**, a mixed-initiative constraint-based planner which aids operational robotic field experimentation by integrating **EUROPA** and **NEPTUS**.

As opposed to the previously presented centralized planner (section 3.1), **EUROPTUS** does not attempt to model all constraints comprehensively, but is used primarily for task assignment, information gathering and situational awareness of a network of vehicles based on a high-level description of how the entire system should evolve.

While the abstraction through timelines in **T-REX** allows for efficient modular design and is a good approach to distribute the planning problem, it is not necessary for **EUROPTUS** considering it can run at the base station on much more powerful computers than those embedded on the vehicles.

Moreover, in an autonomous systems such as **T-REX** , planning and execution are intertwined. Both manipulate the same plan representation and plan execution is required to occur at every clock cycle (synchronization step). Such design, while appropriate for embedded systems, is

incompatible when the execution is distributed among a network of vehicles and coordination is done via limited communication channels with varying latencies and availability.

3.3.1 Tracking Execution within Deliberation

For **EUROPTUS**, computing power is not a critical issue but communication limitations and observational updates is. **EUROPTUS** needs to account for the fact that world evolution will often be observed later than occurrence, if at all. However, execution can still be integrated into a deliberation process as planning and we go a step further in considering that execution is a part of planning.

Execution feedback is integrated into the the deliberation process by taking full advantage of the way the planner works. Specifically, the planner will stop searching as soon as no more flaws are found; and the introduction of a new token (including *Facts*) in the plan create new flaws in the plan.

Let's consider that the planner has no more flaws and that the next command to be executed has been already dispatched. Because of communication disruption, we receive feedback from AUV_a that indicates that its state changed from *Inactive* to *Operating* an hour ago. Our approach is then to create the *Fact* token *Operating* for AUV_a starting at the time corresponding to an hour ago. This token is added to the plan generating a new *Open condition* flaw that the solver needs to resolve.

The resolution can be either as simple as a merge, if the token reflects exactly what was planned, or it is conflicting with the partial plan requiring in turn, the planner to backtrack. The insertion of such a token as a *Fact* however, is akin to a plan recognition in the context of the current plan maintained. As the planner operates continuously keeping its search state alive, this recognition can impact the search by forcing the planner to backtrack over past decisions until it finds an alternative new solution given the injected Fact.

As long as we assume that the decisions impacted by the new observation are not close to the root of the search tree it allows for a plan resolution in few steps without an adverse impact to performance. Further, such an assumption is reasonable as often past observations arrive in relative chronological order rarely impacting the distant past in plan history. These steps also highlight how execution tracking can be a pure deliberative task within the same planning engine.

The remaining problem is to decide which part of the plan is ready to be dispatched for execution. Many actions in a partial plan can be conditioned by the need to observe a situation. In **EUROPA**, an action is a special kind of token with temporal relations expressed either as *conditions* necessary for its execution, or expected *effects* of this actions [RPB12].

3.3.2 Multi-Asset Domain Model

The developed planning domain model aims to coordinate the operation of multiple multiple vehicles towards synchronous surveys on the same area. This can be used for obtaining synoptic observations (air, surface and underwater) of natural phenomena. More specifically, we devised a

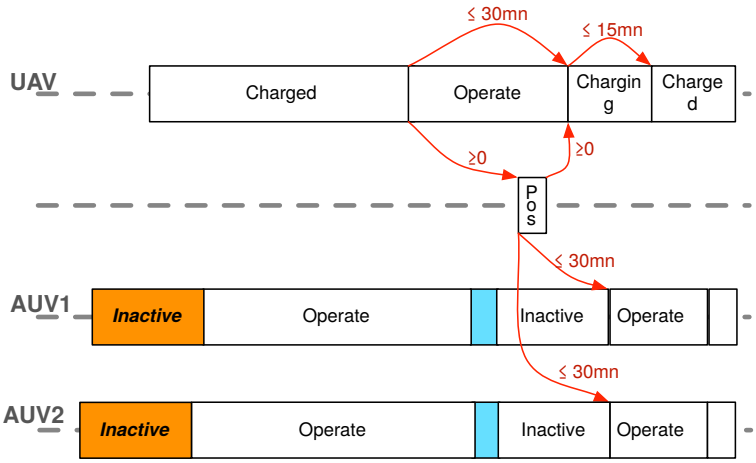

Figure 3.12: Simplified domain model used by **EUROPTUS**

model where 2 AUVs perform coordinated surveys around a point of interest that may have been previously detected by an UAV.

To command 2 AUVs for a coordinated survey **EUROPTUS** is required to observe that both AUVs are ready and that there is a "fresh" position of interest detected by an UAV. When deciding on dispatching a partial plan for an AUV survey, **EUROPTUS** does an analysis of the causality structure of an action with its tokens.

In a complete plan, an action in the plan can be dispatched for execution when it is considered *Justified*, otherwise if becomes a *Pending action* flaw, as per the following definitions:

Justified action An action is *Justified* if either it is a *Fact*, the condition of a *Justified* action or it is an action for which all the effects are *Justified*.

Pending Action An action that could start at the current time but does not have all of its conditions *Justified* nor is it *Justified* itself. Its default resolution is to restrict the start time to be postponed.

While action justification is reasonable, we need to ensure that we do not dispatch the action before its valid start time. However, we could potentially be in a situation where an action does not have all its justified conditions due to one or more missing messages and yet its start time is before the current time (i.e. it *should have* been dispatched for execution in the past). Dispatch is then postponed and we address this as a *Pending action*.

Consequently the planner needs to postpone dispatching this action until either new observations justify the action or the start time can no longer be pushed; the latter triggers a backtrack for an alternate solution.

Any pending action that has all of its conditions *Justified* is dispatched to **NEPTUS** for execution which eventually will receive the observation of its completion (from the vehicle) and report it to **EUROPTUS**. This results in a control loop that is managed as a pure continuous deliberation process governed by the principles just described.

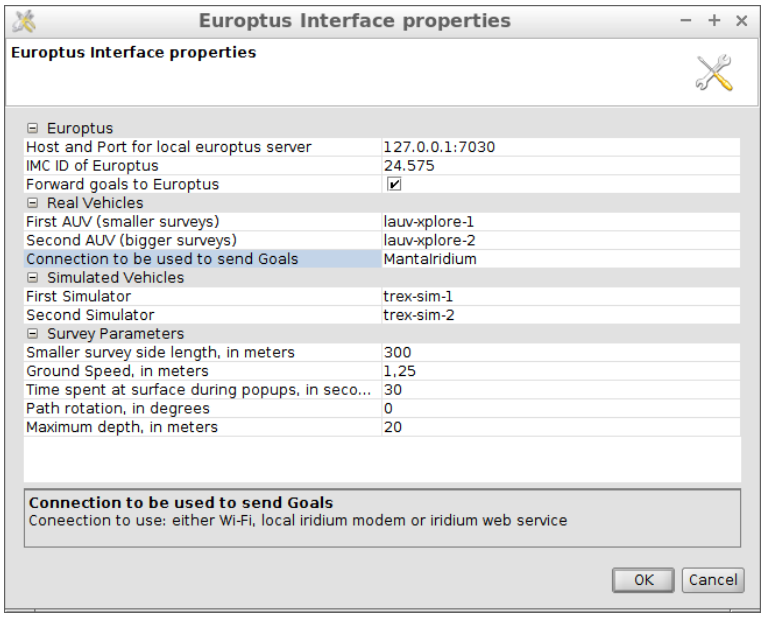

Figure 3.13: **NEPTUS** plug-in properties for **EUROPTUS** interaction

3.3.3 Neptus Infrastructure for Distributed Planning

A cornerstone of multi-vehicle operations, is the communications infrastructure. This development leveraged the existing LSTS toolchain and Neptus infrastructure in order to interface multiple vehicles deployed in the field. All communications between **NEPTUS** and the vehicles are done using the **IMC** protocol. This allows the information to flow over different transports such as Wi-Fi, Iridium (satellite) or acoustic modem.

In **NEPTUS**, a new plug-in was created to abstract all different communication means and fuse information coming from any one of them. **NEPTUS** interacts with **EUROPTUS** also using **IMC** . Moreover, the same **NEPTUS** plug-in also maintains a list of shore-side **DUNE/T-REX** simulators which are used to improve the operators' situational awareness of vehicle positions and progress.

We used **DUNE/T-REX** simulated vehicles running off the commands generated by **EUROPTUS** and sent to the actual platforms. As a consequence, the simulated vehicles execute the identical set of commands albeit in an idealized environment. While doing so and in periods of loss of contact, operators can determine with clarity, what each asset is expected to be doing. In the case of AUVs, any updates received over any available communication channel, are used to reset the simulated vehicle's localization filter.

For commanding the vehicles, the **NEPTUS** plug-in has a set of configurable parameters (Figure 3.13) that are used to define how to send the commands to the remote vehicles (and whether to use local simulators).

Vehicle operations must ingest the data coming from multiple sources and allow variable latency for incoming data and outgoing commands. For aggregating data from multiple sources we also developed a centralized communications "Hub" developed using Google App Engine. This cloud-based application accepts data pushed through Iridium, Globalstar and Argos satellite

communications, as well as several web entry points for posting real-time and/or historical information. This is what the **NEPTUS** plug-in uses to maintain connectivity with the vehicles when it has web access, otherwise it needs to use a local Iridium modem for sending and receiving all data.

Chapter 4

Results

4.1 Multi-Vehicle Planning Tests

In this section we report a field deployment where 3 bottom mapping AUVs with heterogeneous sensing payload were tasked using the centralized planning approach described in section 3.1 [CPR+15].

4.1.1 Motivation

For sea-floor surveys, AUVs typically need to travel while maintaining a constant altitude over ground. This is primarily because for each sonar range and frequency configuration, there is an optimal distance at which the seafloor can be sampled to derive the appropriate resolution of the bathymetry. As a result, planned trajectories must be in accordance to the selected side-scan sonar configurations.

Mine-hunting operations are usually executed with a single AUV or using several homogeneous vehicles as plans created manually by operators would need to be adapted to individual vehicle hardware configurations, periodicity of surfacing, side-scan sonar configurations, planned depths, etc.

Conceptually, the use of heterogeneous vehicles for such an application reduces the overall operation time, because some of the vehicles may be better suited for surveying large areas while others may have higher resolution sensors and/or navigation in order to take high-resolution images of detected contacts. However this results in a high cognitive burden on the operator. Moreover, manual planning for AUVs also requires a number of safety procedures that can add to operator overload when using multiple heterogeneous vehicles. Such a scenario is therefore ripe for our evaluation using automated planning.

4.1.2 Field deployment

For evaluation, we used 3 LAUV vehicles in a mine-hunting scenario where they are equipped with side-scan sonars to image the sea floor, an acoustic altimeter and a high-resolution camera.

Figure 4.1: Light Autonomous Underwater Vehicles used for this test.

After side-scan images are inspected by operators, a set of objects of interest or *contacts* and their locations are determined and the AUVs are dispatched to identify those objects with high resolution cameras to be ground-truthed.

We used 3 LAUV's (Noptilus-1, -2 and -3) in our experiments (depicted in 4.1). All vehicles have different payload configurations but similar navigation accuracy using the same Inertial Navigation System (INS). Noptilus-1 and Noptilus-3 carry lower-resolution Imagenex side-scan sonars and Imagenex DeltaT Multibeam sensors. Noptilus-3 carries an underwater high-resolution cameras and Noptilus-2 carries only an Edgetech side-scan sonar. Noptilus-1 carries an RBR CTD (conductivity/temperature/depth) probe while the other vehicles use instead a sound velocity sensor for correcting sonar measurements.

The deployment area was inside the Leixões harbor in Porto (Fig. 4.2). The base station was located on top of a pier in the harbor with easy access to the water. The vehicles were deployed from near the base station and tele-operated to the operational area shown in the figure. Inside the operational area, the vehicles were placed in known and safe positions at the surface, which we call *depots*, where communications with the base station was viable. The experiment was then divided in two phases. In the first phase the vehicles were used to survey the operational area while in the second they were used to reinspect and identify points of interest detected in the data acquired earlier [1].

In phase one, the operator defined a set of areas of interest to be surveyed using side-scan or multibeam sonar sensors. Fig. 4.3 is a **NEPTUS** console view taken during the experiment after the operator specified the survey area. In order to task the vehicles, the operator requested **NEPTUS** to generate the plans for the vehicles which results in translating the current world state to `PDDL` and generating a set of solutions using **LPG-TD**. The best solution is selected

[1] https://www.youtube.com/watch?v=qWHXRek8_so

Figure 4.2: The deployment area, Leixões harbor, Porto.

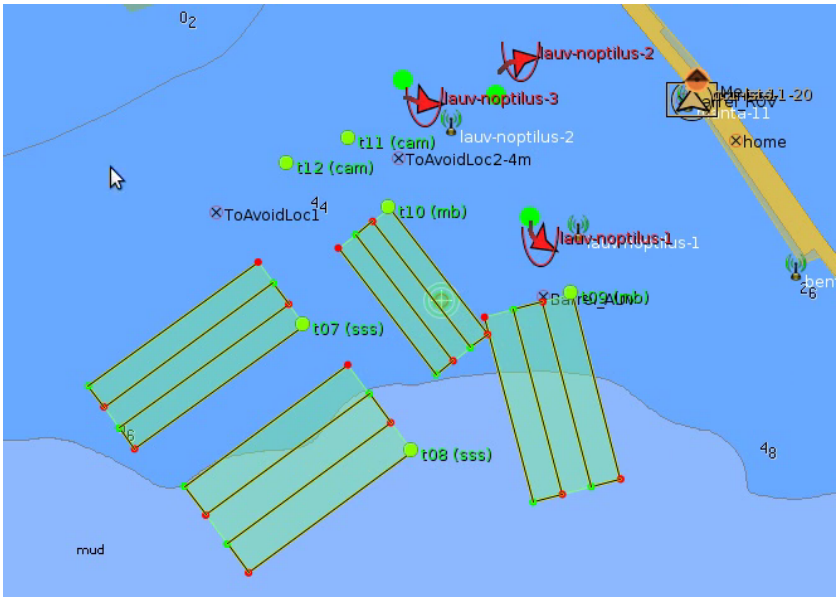

Figure 4.3: Detail of depots and objectives from Fig. 4.2 defined for phase one of the experiment.

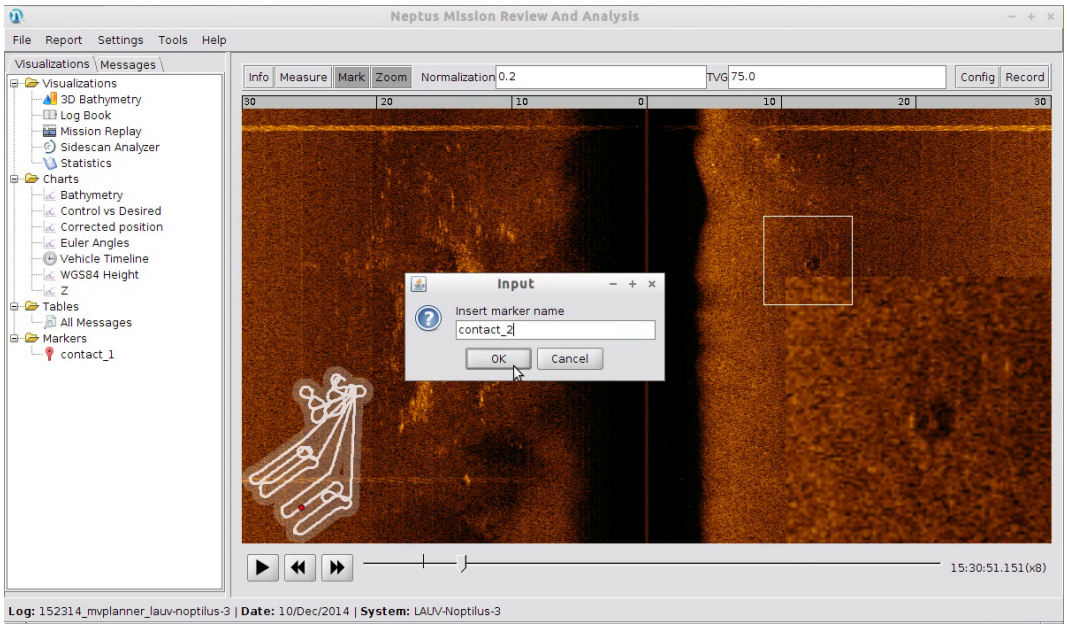

Figure 4.4: Side-scan data analysis and addition of contact in **NEPTUS**.

automatically based on overall execution time. As a result a set of scripted plans is generated and available for the operator to visualize and simulate. The operator can visually verify the plan and change the objectives and regenerate a solution accordingly if needed.

When the operator has validated the generated plans, s/he orders its execution by all assigned vehicles. After the vehicles perform the initial surveys, all side-scan data is downloaded to the base station and inspected by the operator in determining contacts in the sampled regions as seen in figure 4.4.

In phase two, a new set of objectives is then defined by the operator in order to get a closer view of these potential contacts. For some contacts it is likely that the operator not only requests a camera inspection but also CTD sampling in order to augment identification. The deployment ends when all vehicles return to their respective depots after completing all objectives.

4.1.3 Experimental Data and Analysis

Fig. 4.5 shows a visualization of side-scan data (in brown) that was acquired on the first phase of the experiment showing complete coverage of the survey area. Moreover, the contacts identified by the operator at the end of this phase are indicated with yellow markers. In the second phase of the experiment, these contacts were visited by the vehicles and the tracks performed by the vehicles in phase two are overlayed within **NEPTUS**.

The behavior of the vehicles mapped well into the specification of the operator; however the action durations had discrepancies compared to the predicted plan times. Table 4.1 shows the average difference between predicted and actual times for executing different actions. The results show that executing *communicate* and *sample* actions is more accurate than *survey* and *move*.

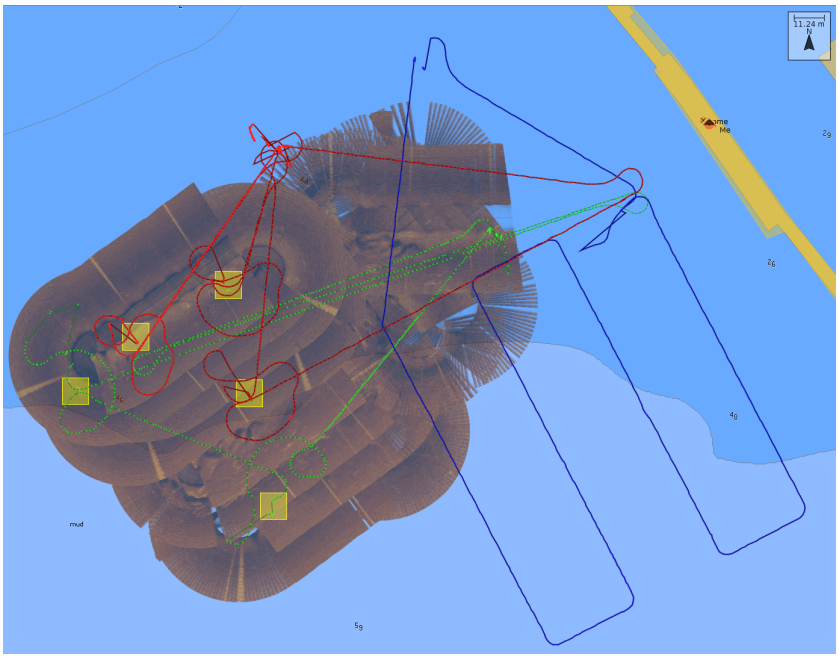

Figure 4.5: Vehicle trajectory in the second phase of the experiment, overlayed on top of side-scan data (brown) and identified contacts (yellow).

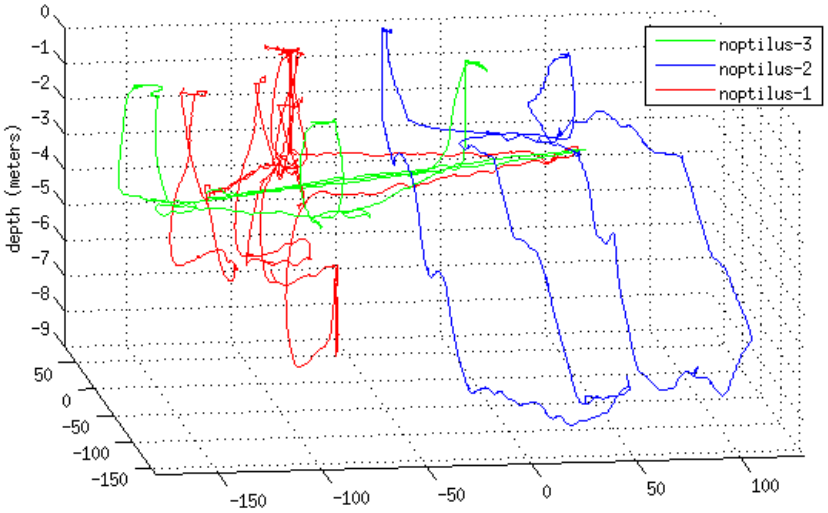

Figure 4.6: 3D view of the vehicle trajectories for the second phase of the experiment with trajectories from three AUVs.

Vehicle	Action	Average Time Difference	Standard deviation
Noptilus-1	move	47,80	49,11
	survey	23,15	23,26
	sample	1,33	0,58
	communicate	0,16	0,17
Noptilus-2	move	39,57	35,66
	survey	107,88	141,10
	sample	N/A	N/A
	communicate	0,25	0,07
Noptilus-3	move	59,90	57,05
	survey	24	0,00
	sample	9,57	13,64
	communicate	0,11	0,16

Table 4.1: Difference between planned and execution times.

This occurs because *communicate* and *sample* in particular, generate timed actions while the other actions generate a behavior that requires the vehicle to move between different locations whose completion time depends on environment conditions such as water currents, sea floor roughness and time to achieve depth. The roughness of the terrain can be seen in Fig. 4.6 where the trajectories of the vehicles are plotted in 3D. To give an example, the vehicle takes longer to reach the depth required to take a camera image if the location is in a deeper point of the operational area and this depth is not modeled a priori in the planner.

For movement actions, **DUNE** also appended a surfacing behavior. This was added in order to re-acquire GPS positions and therefore improve the quality of localization for upcoming samples and surveys. This is important, since it may require the vehicle to stay underwater for a substantial period of time with consequent localization errors. However this was not reflected in the planning domain model. Since the generated plans do not require time coordination of the vehicles, this did not impact the end result from the operator perspective.

4.2 Sunfish Tracking Experiment

This section documents the Sunfish Tracking Experiment where scientists from different institutions and backgrounds joined efforts in order to study the behavior of ocean sunfish. Our approach was to use satellite markers tied to the fish and coordinating multiple assets in order to survey the fish from the air and its surounding water column.

4.2.1 Motivation

Recent studies such as [SQH+09] have related the perceived positions of Sunfish with a foraging pattern of resource “hot-spots”. These resources are consumed not only by Ocean Sunfish but potentially many other species. As a result, Sunfish locations are expected to be an indicator of an healthy food chain including other fish and respective predators.

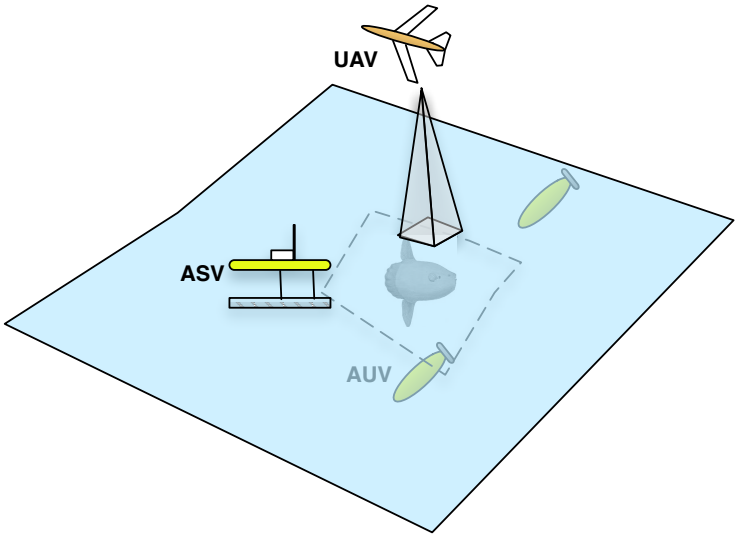

Figure 4.7: Concept deployment for this experiment.

Sunfish is the largest bony fish with an average adult weight of 1000 kg. They travel long distances and tens of specimens are captured everyday off the Portuguese coast and, as the fish is not edible, they are released back to the water by fishermen. Sunfishes dive down to 600 meters of depth but often stay next to the surface for several minutes or even hours at a time. However, they travel large distances and scientists are interested not only in the samples provided by fish sensors but in synoptic observations where fish position and all the area surrounding the fish is completely mapped as quickly as possible. This is feasible only with multi-asset deployments and adaptive sampling [FLB+06] techniques: ships, robots and sensor positions are adapted whenever a new focus of interest (fish) is detected.

Ocean Sunfish properties and their typical behavior makes it a perfect target for a tracking experiment. These periodic "pop-ups" can simplify the process of tracking this fish, by tagging them with satellite markers that report the positions while at surface. Whenever a sunfish position is reported, an UAV can be sent to fly over the fish tag and, if spotted, the AUV is sent to to survey the area (Figure 4.7).

4.2.2 Hardware Overview

In order to track and map the surroundings of fish in the open sea, several new hardware technologies were developed/used: low-cost satellite markers, autonomous underwater vehicles, unmanned aerial vehicles and communication gateways.

4.2.2.1 Satellite fish markers

The fish markers' objective is to be tied onto a fish and to send its position periodically while at surface. Incomplete knowledge of the sunfish behavior led us to test two different satellite local-

Figure 4.8: Argos (left) and SPOT (right) satellite markers used in the sunfish experiment.

	Spot Tag	**Argos Tag**
Service	Globalstar	Argos
Production cost	300	1000
Time to send position	3 minutes	10 seconds
Position error	$<$ 5 meters	$>$ 100 meters
Messages	up to 700	up to 27000
Weight in air	350g	150g

Table 4.2: Satellite tags comparison

Figure 4.9: Some of the hardware used in the experiment: LAUV (top-left), X-8 UAV (bottom-left) and Manta (right).

ization providers. The Argos system[2] has already been tested numerous times for animal tracking and allows fast localization but has higher positioning errors and uncertainty (see Table 4.2). In a more ambitious approach, we tested the low cost FindMeSpot[3] service which provides an higher position accuracy with the drawback of requiring more time for initial localization (immediately after the fish comes to the surface) and higher battery consumption (GPS acquisition).

Both tag models were designed taking into account the targeted fish size which constrained the maximum admissible buoyancy to 20g and allowing a maximum operation depth of 600m. A manufacturing safety margin had to be ensured due to the fact that the tags were handmade (in-house prototypes) and therefore the manufacturing process will inherently lack precision. A total of 20 small-sized sunfish were tagged and 15 of them periodically reported their position back.

4.2.2.2 Autonomous Underwater Vehicles

The LAUV (Light Autonomous Underwater Vehicle) is a lightweight, modular platform capable of carrying a set of different sensors. It is specially designed to be a highly operational and cost-effective surveying tool for oceanographic, hydrographic and environmental surveys. The model used in the experiment is equipped with a calibrated XR-620 CTD sensor from RBR Ltd that registers water conductivity, temperature and depth accurately. Additionally a GoPro camera was mounted in the front of the AUV.

The type of application of the vehicles used in this experiment is very different of those in section 4.1 and, as such, we used a different vehicle class more directed towards oceanography. Together with the sensor payload which in this case are strictly environmental probes, the inertial

[2] http://www.argos-system.org/
[3] http://findmespot.eu/

sensor is less accurate but the vehicle can operate for much longer periods of time, allowing up to 24 hours of continuous operation.

For this test, in addition to its main CPU (running **DUNE**), an auxiliary CPU was added in order to run **T-REX**. An Iridium modem and respective antenna was added allowing LAUV to be controlled via satellite event when there is no Wi-Fi or GSM coverage. Also, in order to achieve 24 hours of continuous operation at 1 m/s, LAUV was equipped with extra battery packs which made up a total of 1500 Wh.

4.2.2.3 Unmanned Aerial Vehicles

The X-8 is a low-cost delta wing UAV which is easy to launch and quick to recover. The used frame is commercially available and adapted in the LSTS where it is enhanced with a communications and computational stack that allows it to be part of a multi-vehicle network. These UAVs have already been used for quick surveillance missions, low altitude reconnaissance with live video feed and data muling between remote locations.

For this test, a X-8 UAV was equipped with an infrared imaging camera which allowed scientists to perceive, in real-time the surface water temperature in mixing regions such as fronts and plumes.

4.2.2.4 Communications Gateway

Mantas connect all different hardware platforms to the existing Ground Stations and are used to receive/publish data to/from the web.

The Manta Gateway is a portable communications gateway supporting different communication means such as Iridium, GSM/HDSPA, Acoustic Modem and Wi-Fi. Typically, these gateways are used at or near base stations to provide communication links to the deployed unmanned vehicles. Mantas are also used for improving the operators' situational awareness by disseminating their GPS position and the positions of nearby ships (received via their AIS receiver). The Wi-Fi network can also be configured to run in *Wireless Distribution System* mode which allows using several Mantas to create a mesh network.

4.2.3 Communications

The information flow between the different components is depicted in Fig. 4.10. We can observe that all information coming from different sources is concentrated in two main systems: Manta gateways for local networks and the **Ripples** for large-scale operations. Moreover much of the systems deployed in the field are capable of understanding **IMC**.

Local network communications are done via the Manta using the **IMC** protocol. This allows reception of real-time telemetry and controlling the deployed assets through scripted plans or high-level objectives. The Manta is capable of compressing / decompressing IMC messages according to the communication mean in use. For instance, if one needs to send a large plan definition to a vehicle using Iridium, the definition may have to be compressed and split into several small

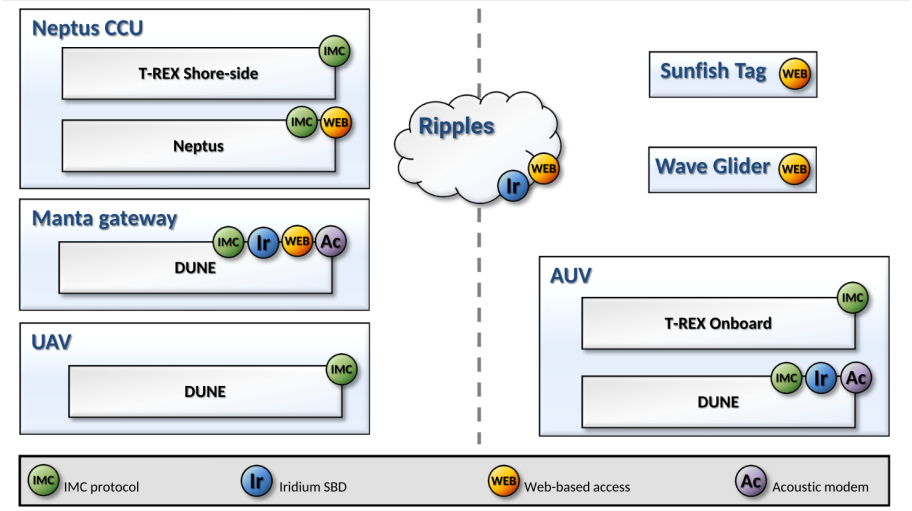

Figure 4.10: Connectivity between different systems during the experiment.

messages (maximum message size of 256 bytes). Something similar happens while using acoustic communications.

Manta provides a connection between the local network and the **Ripples** web service either by using a 3G internet dongle (and using the **Ripples** Web API) or via Iridium messages which are intercepted by the **Ripples**. In order to receive state updates without an Internet connection, **Ripples** provides a mechanism where an Iridium terminal can request for updates over a period of time (or until the terminal unsubscribes in a similar fashion). This mechanism allows operators in remote areas to receive asset positions and to receive any plans being executed by the vehicles.

4.2.4 Operator Interfaces and Situation Awareness

The Sunfish experiment required the deployment of multiple assets spread out in the field (land and ocean). All assets, including consoles, ships, vehicles and satellite tags are sources of information. As seen in Fig. 4.10, we aggregated all information in the **Ripples** repository. The **Ripples** provides a series of Web APIs for pushing and querying state updates and also for transmitting messages to Iridium destinations (relay between Web and Iridium).

Whenever information is sent via the **Ripples**, it gets stored on the **Ripples**. With this in mind, we created a set of visualizations for the information being stored. A simple visualization is provided through a web page. This web page connects to the **Ripples** and provides real-time information updates. A snapshot of this page can be seen in figure 4.11.

The **NEPTUS** user interface is depicted in 4.12. This snapshot shows several additional plug-ins that were developed specifically for this experiment. There is one extra "situational awareness" layer that depicts all fish positions as they are received via Web or Iridium. Not only the present position can be seen but also past ones in order to help scientists predict future positions and where to direct the assets.

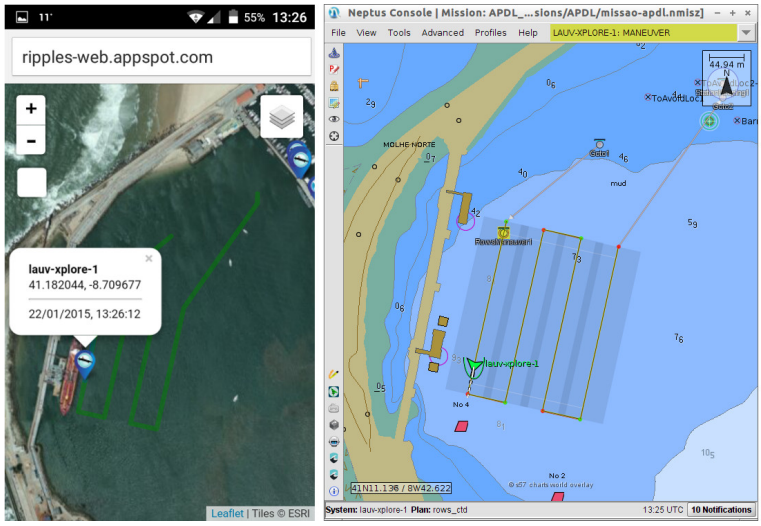

Figure 4.11: Snapshot of **Ripples** web page viewed in a mobile phone (left) and the same information displayed in **NEPTUS** (right)

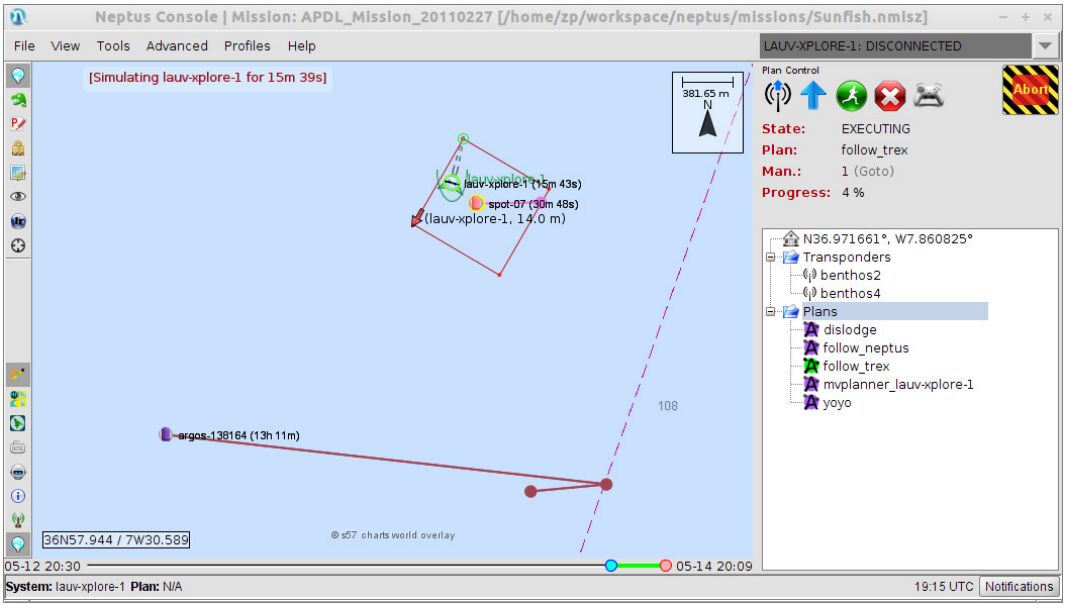

Figure 4.12: Snapshot of the **NEPTUS** interface used for situation awareness and commanding the vehicles

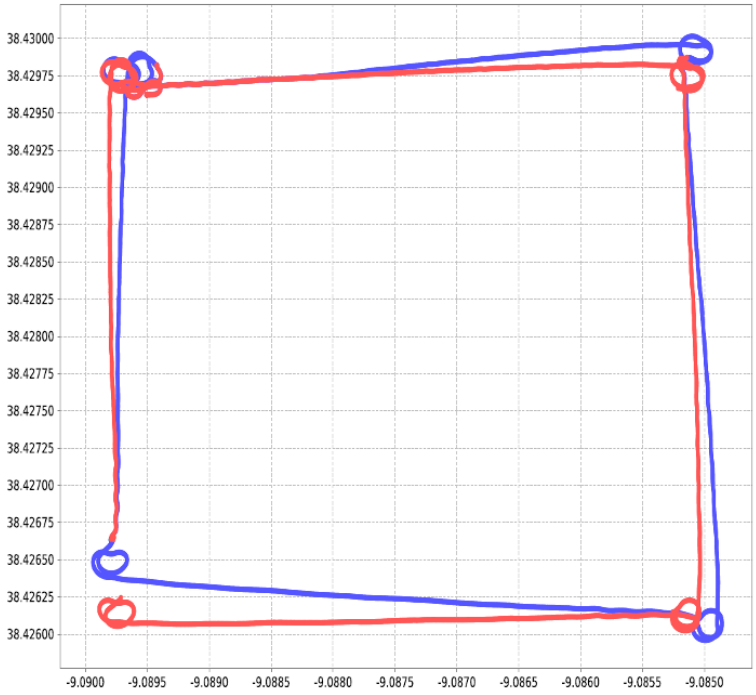

Figure 4.13: Top-view over a survey behavior generated onboard the AUVs by **T-REX** . Red trace is the estimated (uncorrected) track and the blue track is corrected using the GPS positions at the corners.

4.2.5 Mixed-Initiative Planning of AUVs

For the Sunfish experiment, a new **T-REX** interaction mode was added to **NEPTUS** (as a plug-in). This interaction allows the user to select a point or object in the map and request a survey on that object. Whenever a fish is detected and its position validated, **NEPTUS** was this way used to send a survey objective to **T-REX** , irrespective of the current state of the vehicle. As a result, the operator could request one survey at a time or several in a row if multiple points of interest existed.

Under the hood, Whenever a survey is requested by the operator, **NEPTUS** generates an **IMC** message with a **T-REX** goal and sends it to **T-REX** either via Wi-Fi or Iridium communications (through the **Ripples**) as described in 3.2. The used domain model requests the vehicle to do a square survey around the received points and resurface on every corner to improve navigation accuracy. This goal introduces a new "flaw" in the plan which requires the onboard planner to generate a new plan that encompasses a survey around this new location.

The plot in Figure 4.13 shows a survey that was generated and executed around an actual sunfish position during the experiment. In the figure we can also see the localization error the AUV is prone to when travelling underwater and the need for pop-ups at the survey corners. While travelling between corners the AUV was also measuring the entire water column (YoYo reactor) as can be seen in 3D on Figure 4.14.

The scientific results of this experiment are not the subject of this thesis and thus are not

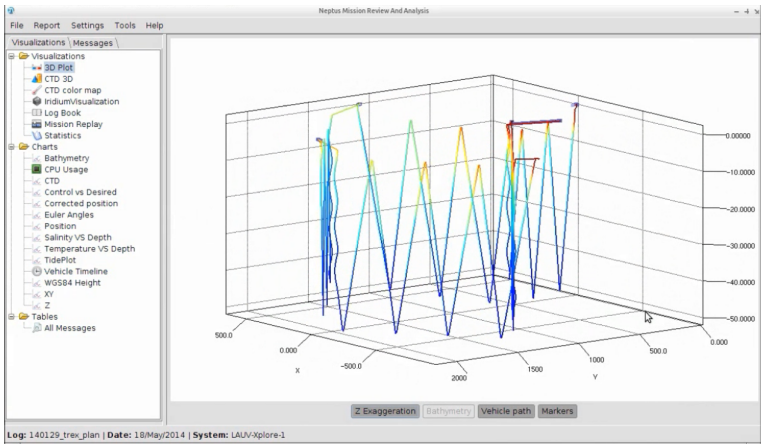

Figure 4.14: 3D view of the survey behavior generated onboard the AUVs by **T-REX**

reported here. However, a scientific article has resulted from this experiment and its obtained data [SLCG+16].

4.3 Recognized Environmental Picture 2015 Exercise

In July 2015, the Underwater Systems and Technology Laboratory undertook a large scale exercise in the Azores Islands in order to detect and track cetaceans using vehicle networks. This section describes the exercise objectives, used hardware and results from field deployments onboard NRP Gago Coutinho.

4.3.1 Motivation

Every year in July the LSTS organizes, together with the Portuguese Navy, the REP exercise where multiple entities developing or interested in the development of new concepts of operation involving autonomous vehicles collaborate and do large-scale deployments of vehicles. REP15 (2015) took place off Faial island in Azores. For that year, one of the objectives proposed by Department of Oceanography and Fisheries from Azores University (DOP-Açores) was to use multiple vehicles to do synoptic observations around sperm whales.

Sperm whales are often spotted nearby Faial islands where they usually spend large parts of the year. Moreover, authors have correlated existance of large groups of whales with underwater sea mounts which are quite common in the portuguese economic exclusive zone [MMK+08]. The reasoning behind this is that around seamounts large quantities of nutrients are raised from deep sea and closer to the surface which generate abundant ecosystems.

The concept of operations would build upon what was done in 4.2, namely using UAVs for detecting sperm whales and use AUVs in-situ to measure the water column around sperm whales. All tests needed to be conducted continuously onboard a navy vessel because the area of interest was in open waters south of Pico Island in the mid-Atlantic archipelago of the Azores in Portuguese waters off the São Mateus Bank (Figure 4.15).

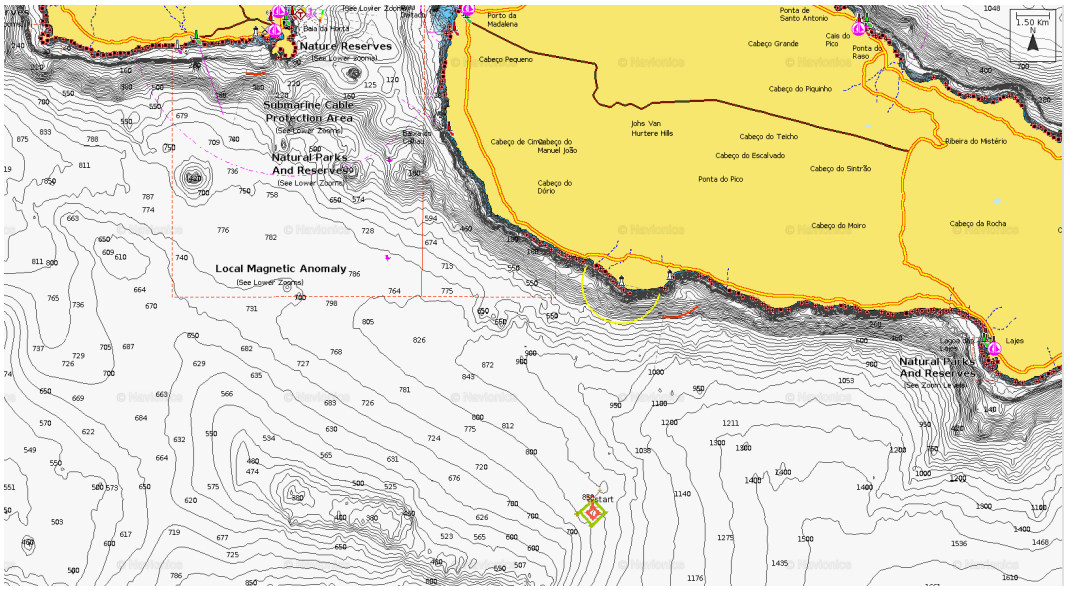

Figure 4.15: Location of the REP15 whale tracking experiment.

4.3.2 Challenges and Approach

Due to the size of the animals and how fast they move the search and survey areas needed to be increased considerable comparing to the sunfish experiments described earlier. In this case all AUV operations needed to be done over satellite communications and coordinated with UAVs operated over Wi-Fi in order to receive live video feed.

The UAVs flown from the NRP Gago Coutinho, carried visible light and infrared cameras that were followed by multiple operators onboard the ship in order to detect animals close to the water surface. Two UAV teams were operating alternately since they had to share the same infrastructure for taking off (catapult) and landing (net). AUVs were deployed in parallel with the UAV operations so that they would be ready to survey any targets detected by the human operators.

From very early in the preparation, we considered one of the main challenges the coordination of all assets according to scientific needs. Not only the robots deployed in the air and water needed to be orchestrated but also the movement of the ship and other smaller boats (RHIBs) used for collecting samples and intervene in case of emergency.

Our approach has been to automate as much as possible the entire operation: define the scientific procedure as a temporal planning domain model and use an automated planner to generate the commands to send to the vehicles and operators during the experiment. This was done using the framework described in 3.3, having onboard planners on two AUVs deployed simultaneuously to the water and also on the ship in order to coordinate the entire experiment.

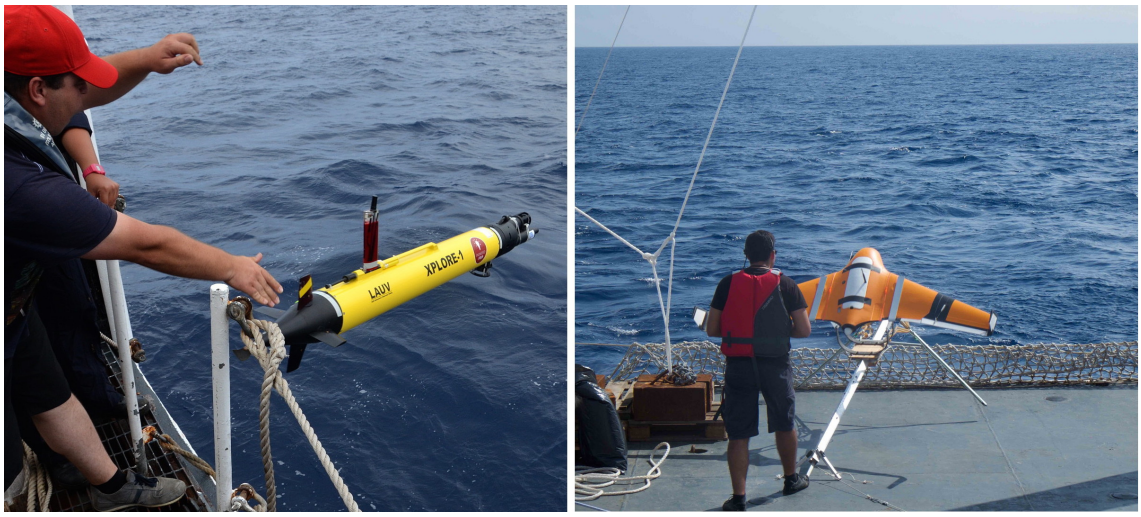

Figure 4.16: Launching of autonomous vehicles onboard NRP Gago Coutinho. LAUV Xplore 1 on the left and Skywalker X-8 05 on the right.

4.3.3 Hardware and Communications Setup

The Portuguese Navy research vessel, the NRP *Almirante Gago Coutinho*[4] was a large vessel used for launching AUVs and UAVs from the aft deck (Figure 4.16). The upper water-column Light AUVs were equipped with RBR *XR*620 CTD (Conductivity, temperature and density) probe, WHOI acoustic modems, an Iridium SBD modem for satellite communication and a Turner Designs Cyclops-7 wet-probe with a fluorometer. There were multiple Skywalker X-8 UAVs at our disposal; two with Far-IR cameras and one vehicle with a new light-weight hyper-spectral imager. Moreover, a multibeam sonar mounted on NRP Gago Coutinho was also used to obtain an high precision relief of the sea bottom.

All operators and pilots were onboard the vessel which had multiple (sectorial and omnidirectional) antennas for providing 802.11 WiFi and Airmax coverage via ubiquiti radios. The antennas were connected to multiple "Manta" communications gateways which along with unmanned vehicles and **NEPTUS** consoles were connected to a local area network onboard the vessel, allowing operators to be aware of the AUV and UAV operations simultaneously (Fig. 4.17).

In the air, one UAV was deployed during the most part of the experiment carrying different types of cameras. Since the battery endurance of the UAV is limited, the two teams (LSTS and NTNU) would be operating alternately every hour or so. At sea, the two Xplore-class AUVs were deployed early in the day and recovered before the end of the day, spending most of the time disconnected from the ship's Wi-Fi. Operation of these two AUVs was mainly done via satellite communications and an acoustic transponder mounted on the ship was also used to receive telemetry and locate the AUVs (measuring time-of-flight of acoustic pings).

Since most of the time the ship was too far from shore, there was no reliable GSM or Internet connection and all commands needed to be sent to the vehicles via Iridium satellite modems

[4] http://goo.gl/hEVmFU

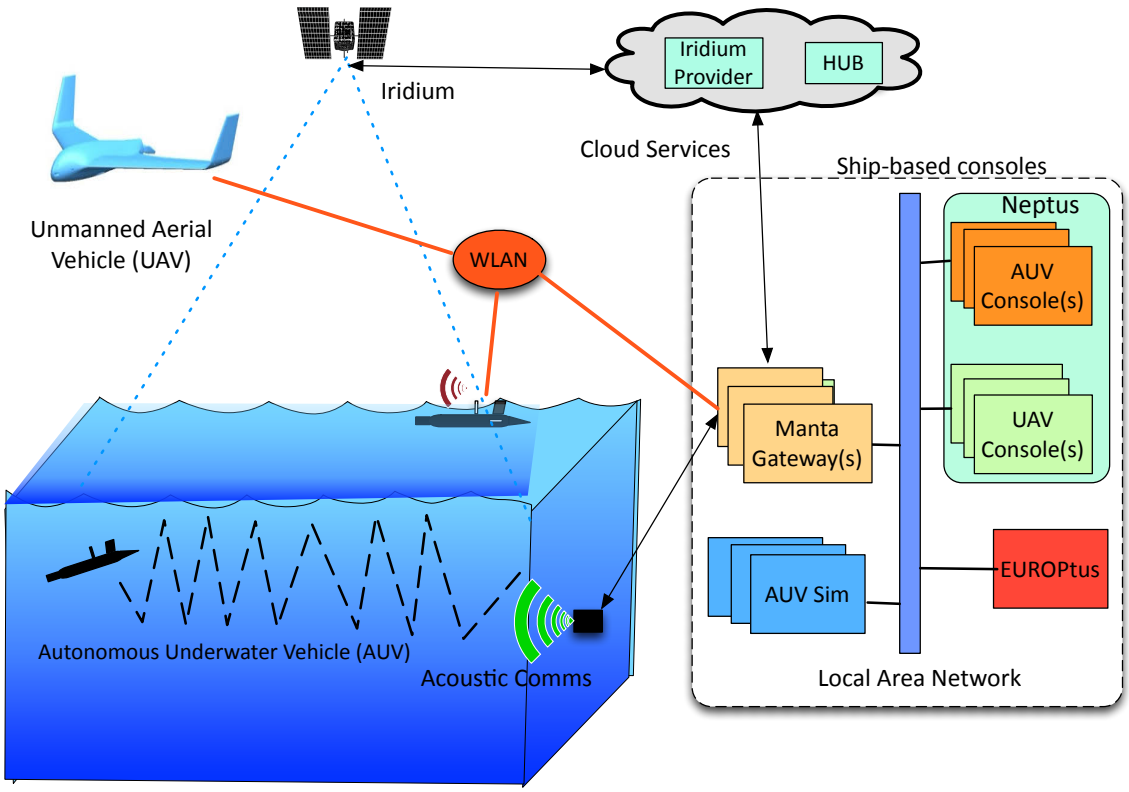

Figure 4.17: Hardware and Communications setup during the REP15 deployment.

mounted on the manta gateways. On the sea, the vehicles also needed to reply to commands using satellite communications, oftentimes being the only viable communication mean.

4.3.4 Mixed-Iniative Planning and Coordination

Since all **NEPTUS** consoles were connected to the same ground segment (using ethernet), all **IMC** data on the ship was seamless shared across all consoles. This was used, for instance, for having multiple operators looking at the video footage coming from the UAVs and looking from whales at the surface. Whenever any of the operators spotted a target at surface, s/he could add a map feature on his/her console and the mark would be disseminated to other consoles.

UAV operations still required a lot of intervention from the human operators and pilots but AUV operation, on the other hand, has been almost completely automated by **EUROPTUS** on this experiment. The **EUROPTUS** software was running alongside the consoles and connected to one specific **NEPTUS** console which was used as its communications relay. **NEPTUS** would forward all disseminated whale detections from any of the connected consoles and also any reported observarions coming from the AUVs deployed at sea.

With the received informatin, **EUROPTUS** continually refined a plan for the ongoing operations, choosing when to send the AUVs to a new survey and around which target as well as request new UAV deployments in order to search for more whale positions. All of this formally according to the defined planning domain model.

While the UAVs were tasked by the human operators to search the area around the ship, the two AUVs were always coordinated in order to perform co-temporal surveys around the selected target. The co-temporal survey required the two AUVs to start and finish roughly at the same time but doing surveys at different distances from the point of interest. For that, we selected to do two rectangular surveys, popping up at surface on the corners and having one of the vehicles doing two smaller square surveys while the other vehicle would do a bigger (roughly twice the size) around it.

The resulting tracks of one such survey can be seen in Figure 4.19. The surveys were measuring the water column by doing a saw-tooth pattern from the water surface down to 50 meters and the bigger square side length was 800 meters.

Also the tracks of a mid-day of operations can be seen in Figure 4.20. In it, the typical search pattern used on the UAVs around the ship is visible as it is possible to see that two consecutive whale positions were selected and targetted by **EUROPTUS**. The time span of this plot corresponds to 4 hours of operation in which the UAV was landed twice. The used ship was capable of maitaining its position accurately (using dynamic positioning).

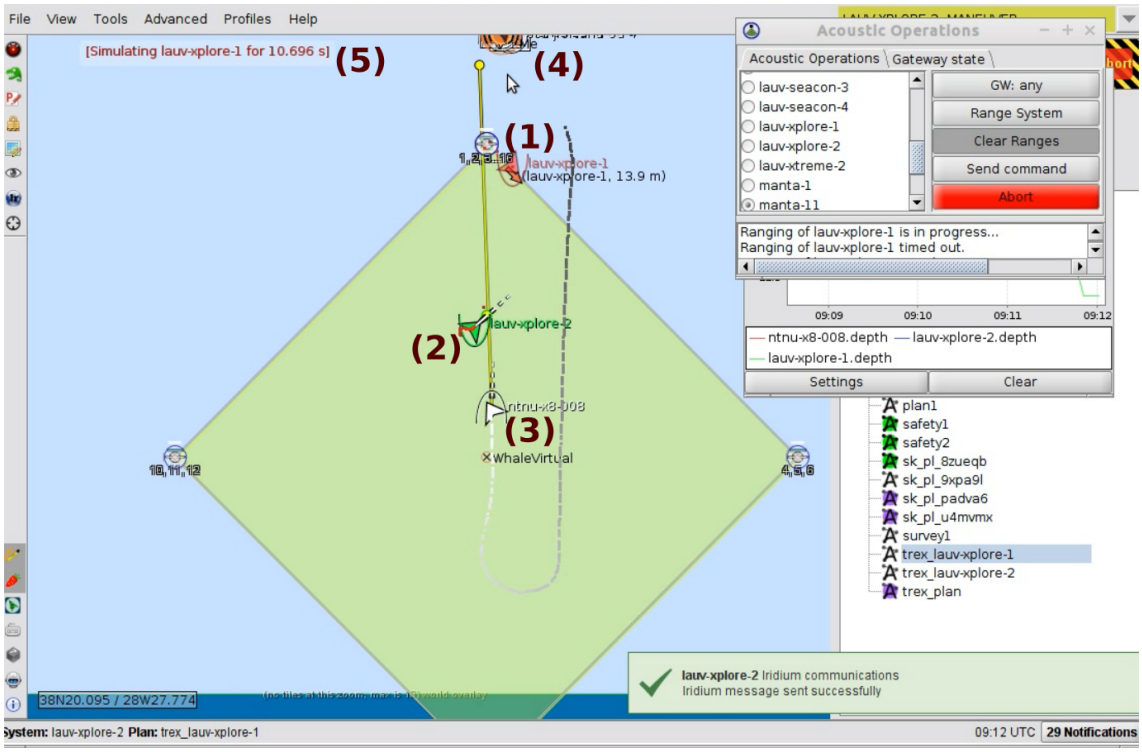

Figure 4.18: **NEPTUS** operator console used as **EUROPTUS** relay.

(1) LAUV Xplore 1 AUV simulated position
(2) LAUV Xplore 2 AUV live position (received via wi-fi)
(3) Skywalker X-008 UAV live position and track (received via wi-fi)
(4) Ground Control Segment (onboard NRP Gago Coutinho) position
(5) Simulating Xplore 1 for 10 seconds (after acoustic position had been received)

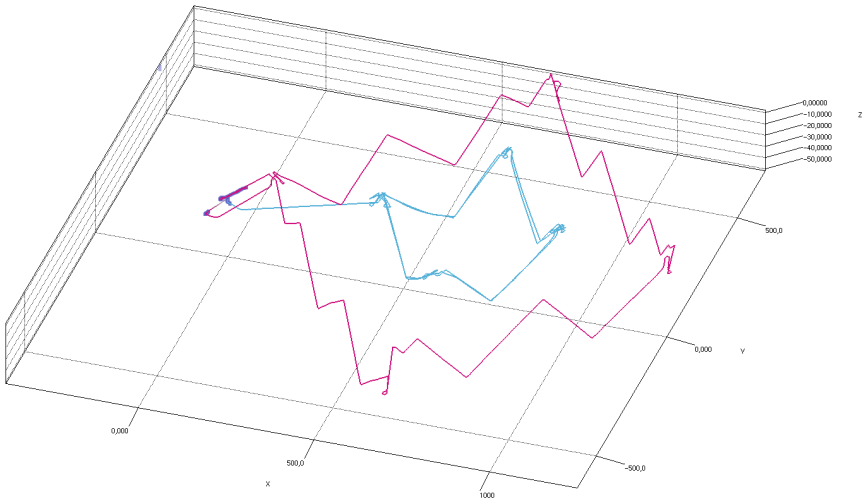

Figure 4.19: 3D tracks performed by two AUVs in one survey commanded by **EUROPTUS**.

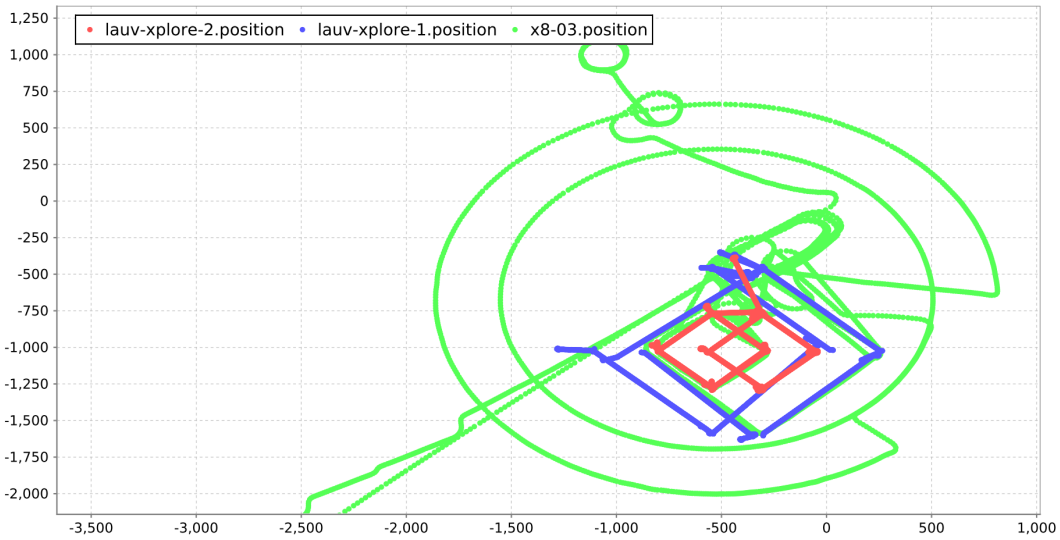

Figure 4.20: Top-side view of the positions of 1 UAV and 2 AUVs operated for a mid-day.

Chapter 5

Conclusions

In this chapter we summarize what was achieved in this thesis and what are its main contributions and shortcomings. Moreover we finish with guiding lines for future work, some of it already ongoing.

5.1 Main Contributions

This thesis has addressed coordination and planning of networks of autonomous vehicles. The work has been very much exploratory in the sense that multiple approaches were tested and validated: centralized planning (3.1), onboard planning (3.2) and distributed planning and coordination (3.3).

5.1.1 Foundational software components

For these developments to be possible, a lot of lacking foundational components were also created such as the **NEPTUS** plan simulation engine, the **Ripples** web service, Iridium communications support in **NEPTUS** and **DUNE** and the **DUNE** *FollowReference* maneuver. Despite being developed for the specific objectives of this thesis, these foundational components have been used a lot outside this scope and are, in itself, one of the main contributions.

The **DUNE** *FollowReference* has been used extensively in deployments as a way to drive vehicles to intended locations from **NEPTUS** and even to tele-operate simultaneously multiple underwater vehicles using acoustic modems.

The developed **NEPTUS** plan simulation tools have been used in many other scenarios for pre-simulation of plans in order to determine the surveyed areas, verification of collision-free plans, verification of line of sight underwater communications throughout plan execution, etc.

The Iridium communications supports not only **T-REX** goals and observations but can also convey any other **IMC** messages which has been used extensively both to receive data from vehicles but also to command regular **DUNE** plans to the vehicles when no other communication means are available.

The **Ripples** web service is currently used whenever third parties need to follow the positions of our vehicles in real-time as well as the positions of surface drifters recently developed by our lab.

5.1.2 Centralized Deliberative Planning

The first approach, where **LPG-TD** was used to generate plans for multiple vehicles is a game changer in the way an operator understands and controls a network of autonomous vehicles. The plans which are generated by the integrated planner are inherently safe in that only valid actions are generated. As a result we can guarantee properties of the system based on the provided domain model such as:

- Only vehicles capable of executing task x are commanded to execute task x;
- Generated plans always finish in a location where there are communications between the vehicle and base station;
- Vehicles are tasked only if they have enough battery to execute the allocated tasks.

Moreover, by using local search, an automated planner can arrive at efficient solutions for allocating multiple tasks to heterogeneous vehicles in very short amounts of time. The same cannot be said of human operators that, while being very capable of controlling one or two homogeneous vehicles at a time usually cannot react on time in virtually any situations more complex than that.

5.1.3 Integration of T-REX Onboard Deliberative Planning

For performing deliberative planning onboard, we opted to use the **T-REX** framework not only because of an ongoing collaboration with MBARI but most importantly because of its very good integration of **EUROPA** which provides a good degree of expressivity while still being capable of generating flexible interval-based plans.

Integration of **T-REX** with **DUNE** and **NEPTUS** has required developing new **IMC** messages for transferring goals and observations between consoles and vehicles. It has also required the development of a new **NEPTUS** plug-in that allows the users to post and recall goals and also synchronizes **NEPTUS** plans with the plans being generated onboard the vehicles by **T-REX**. Moreover, on the **DUNE** side this integration also required developing new **DUNE** behaviors and tasks for controlling **T-REX** execution and also accepting commands via the *FollowReference* maneuver.

The integration of **T-REX** in the LSTS toolchain has also been extended for integrating the MOOS-IVP autonomy framework[1]. This integration was greatly simplified because all underlying required components were already in place and needed only to be adapted.

[1] https://github.com/LSTS/moos-ivp-dune

5.1.4 T-REX-agent for LAUV vehicles

For this thesis, a series of **T-REX** reactors have been developed in order to adapt **T-REX** to the vehicles used by LSTS. Not only new **EUROPA** domain models have been developed but also several new interface reactors that allow safe execution of autonomous behavior by the AUVs.

The developed reactors and domain model has been used in the field in several scenarios and has already been used by other researchers outside of LSTS. Recently, a LAUV vehicle has been operated in the Arctic circle (near Svalbard) running the aforementioned onboard planning components for measuring sea water temperature around icebergs.

5.1.5 Mixed-initiative distributed planning

Finally, one the main contributions of this thesis has been the conception of a distributed planning infrastructure where not only plans are generated on a centralized server but they are also adapted onboard AUVs while enduring harsh communication conditions. Moreover, this infrastructure was tightly integrated with other components in the LSTS toolchain so that it was possible for multiple consoles to exchange information and have deliberative and scripted behaviors running in parallel on the same network.

Finally, instead of allowing the system to run fully autonomous, we improved awareness of the operators by synchronizing the states of shore-side simulators with those of AUVs deployed remotely running onboard planners. This enabled the users not only to better understand the behavior of disconnected vehicles but also intervene in case this behavior is not desired.

5.2 Discussion of Results

The initial (centralized planning) approach despite simplistic is still one the most effective solutions to interface a network of heterogeneous robots and has been validated in the field under a realistic (mine hunting) scenario. This approach, however, can only be employed in situations where there is good communication with all robots and there is no need for replanning throughout the deployment.

For rather static scenarios such as bottom-mapping where the environment is not prone to changes, centralized planning is a very effective approach in that an human operator is capable of efficiently task multiple vehicles given a whole range of constraints and objectives.

On the other hand, if robots are prone to fail during deployment or there is a need to react to dynamic phenomena, centralized planning has its limitations. Since robots can stay disconnected for long periods, they must be capable of autonomously adapt to changes in the environment with no or very limited communications with other systems.

This is the reason why we consider that onboard plan adaptation is so important in networked vehicle systems. If all vehicles rely on a centralized planner to determine their low-level actions they will always be constrained by having a good communication link to the planner whenever there is a need to replan.

Consider for instance that one of the AUVs fails midway while executing a plan commanded by a centralized planner in a place where it has no communication with the base station. The result might be the vehicle entering an undesired state where, despite being able to execute new actions, does not receive any because it has lost connectivity with the planner. This can be overcome by having an onboard planner that, whenever idle and disconnected from base station can actively go to its depot location to receive new actions / goals.

Having all deliberation being done onboard, however, can reduce the awareness of operators because not only the vehicles will remain disconnected for large amounts of time, they will also change its behavior while disconnected.

Instead we envisioned and field-tested an infrastructure where multi-vehicle plans are generated at the base station but these plans are flexible enough that they do not require a very restrictive set of actions from the vehicles but only a set of end states that shall be observed after some time (e.g. finishing a survey within a time window).

Fully autonomous AUV deployments have been conducted using this approach where the centralized planner would synchronize its internal state with the perceived world state and plan future actions according to a scientific global objective. In order to be possible to conduct this type of deployment, however, we needed to allow the operator to understand what was being commanded and potentially intervene if s/he disapproves the generated behaviors. For that, the **NEPTUS** integration included shore-side simulators and automatic translation between **T-REX** actions and **NEPTUS** plans which allowed the operator to effectively supervise the autonomous behavior of the system and intervene as needed.

5.3 Future Work

The work on this thesis has been tested several times and used for real world experiments. However, there still is much to do concerning all 3 main developments of the thesis.

Regarding centralized planning, we are currently starting to work on a new version of the domain model that not only considers the present state of the system but also future states when the vehicles will become available as predicted by the **NEPTUS** simulation engine. The domain model will then be allowed to generate actions for vehicles that will eventually become idle. The execution engine will also need to be changed in that it must verify when to trigger the generated actions as they might be planned for a future state of the system.

Also regarding the centralized planner, it is our intention to make the onboard execution more adaptive to environment changes that cause delays. This can initially be done by using other maneuvers that keep track of time such as *FollowTrajectory* and the new *ScheduledGoto* (waypoint that also includes time reference). Next, we want to generate plans that are executed and adapted onboard by **T-REX**.

The current onboard planning implementation is still a proof-of-concept. The developed domain models are still very simplistic and aimed at open sea scenarios such as oceanography and

biology. In order to be possible to use **T-REX** in more scenarios, we want to create domain models with more expressivity which include bottom-mapping and inter-vehicle communications.

Finally, we want to develop new domain models for networked vehicle systems that can be executed using **EUROPTUS**. The only developed domain model was created according to scientists and had hard requirements concerning communications and survey shapes. However, many others can be devised for practical scenarios that also require coordination of multiple vehicles such as habitat mapping, mine countermeasures, emergency response, among many others.

References

[BCC12] J. Benton, A. J. Coles, and A. Coles. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*, 2012.

[Bru01] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[CPR+15] Lukáš Chrpa, José Pinto, Manuel A Ribeiro, Frédéric Py, Joao Sousa, and Kanna Rajan. On mixed-initiative planning and control for autonomous underwater vehicles. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 1685–1690. IEEE, 2015.

[DFG+05] Paulo Sousa Dias, Sergio Loureiro Fraga, Rui MF Gomes, Gil M Goncalves, Fernando Lobo Pereira, Jose Pinto, and Joao Borges Sousa. NEPTUS- a framework to support multiple vehicle operation. In *Proc Oceans MTS/IEEE Conference*, pages 963–968, Brest, France, 2005.

[dSeTS09a] Laboratório de Sistemas e Tecnologia Subaquática. IMC Online Source Code Repository. http://github.com/LSTS/imc/, 2009. [Online; accessed 1-February-2016].

[dSeTS09b] Laboratório de Sistemas e Tecnologia Subaquática. LSTS source code repository. http://github.com/LSTS/, 2009. [Online; accessed 1-February-2016].

[ECR00] Ben Erwin, Martha Cyr, and Chris Rogers. Lego engineer and robolab: Teaching engineering with labview from kindergarten to graduate school. *International Journal of Engineering Education*, 16(3):181–192, 2000.

[FJ03] J. Frank and A.K. Jónsson. Constraint-based Attribute and Interval Planning. *Constraints*, 8(4):339–364, oct 2003.

[FL03] Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[FLB+06] Edward Fiorelli, Naomi Ehrich Leonard, Pradeep Bhatta, Derek A Paley, Ralf Bachmayer, and David M Fratantoni. Multi-auv control and adaptive sampling in monterey bay. *Oceanic Engineering, IEEE Journal of*, 31(4):935–948, 2006.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[GSS03] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs in LPG. *J. Artif. Intell. Res. (JAIR)*, 20:239–290, 2003.

[Kue13] Johannes Kuehn. ROS code quality, 2013.

[M. 04] M. Ghallab and D. Nau and P. Traverso. *Automated Planning Theory and Practice*. Elsevier Science, 2004.

[Mar06] Eduardo RB Marques. Seaware: A publish/subscribe communications middleware for networked vehicle systems. 2006.

[MDM+09] Ricardo Martins, Paulo Sousa Dias, Eduardo RB Marques, José Pinto, Joao B Sousa, and Femando L Pereira. Imc: A communication protocol for networked vehicles and sensors. In *Oceans 2009-Europe*, pages 1–6. IEEE, 2009.

[MMK+08] Telmo Morato, Miguel Machete, Adrian Kitchingman, Fernando Tempera, Sherman Lai, Gui Menezes, Tony J Pitcher, and Ricardo S Santos. Abundance and distribution of seamounts in the azores. *Marine Ecology Progress Series*, 357:17–21, 2008.

[MPR+08] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. A Deliberative Architecture for AUV Control. In *ICRA*, Pasadena, May 2008.

[New08] Paul Michael Newman. Moos-mission orientated operating suite. *Massachusetts Institute of Technology, Tech. Rep*, 2299(08), 2008.

[NYT01] NYTimes. Sony Tightens Leash on Its Robotic Dog. http://www.nytimes.com/2001/11/05/technology/05AIBO.html, 2001. [Online; accessed 30-July-2015].

[PDG+06] José Pinto, Paulo Sousa Dias, Rui Gonçalves, Gil Manuel Gonçalves, João Tasso de Figueiredo Borges Sousa, Fernando Lobo Pereira, et al. Neptus a framework to support a mission life cycle. In *Proceedings of the 7th Conference on Manoeuvring and Control of Marine Craft*, 2006.

[PRM10] F. Py, K. Rajan, and C. McGann. A Systematic Agent Framework for Situated Autonomous Systems. In *AAMAS*, 2010.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[RPB12] K. Rajan, F. Py, and J. Berreiro. Towards Deliberative Control in Marine Robotics. In M. Seto, editor, *Autonomy in Marine Robots*. Springer Verlag, 2012.

[SB03] François Serra and Jean-Christophe Baillie. Aibo programming using open-r sdk tutorial, 2003.

[SCK+13] Mohammad Munshi Shahin Shah, Lukás Chrpa, Diane E. Kitchin, Thomas Leo McCluskey, and Mauro Vallati. Exploring knowledge engineering strategies in designing and modelling a road traffic accident management domain. In *IJCAI*, 2013.

[Set12] Mae L Seto. *Marine robot autonomy*. Springer Science & Business Media, 2012.

[SLCG+16] Lara L Sousa, Francisco López-Castejón, Javier Gilabert, Paulo Relvas, Ana Couto, Nuno Queiroz, José Pinto, Paulo Sousa Dias, Frederic Py, Margarida Faria, et al. Integrated monitoring of mola mola behaviour in space and time. *PLOS one*, 11(8):e0160404, 2016.

[SQH+09] David W. Sims, Nuno Queiroz, Nicolas E. Humphries, Fernando P. Lima, and Graeme C. Hays. Long-term gps tracking of ocean sunfish mola mola offers a new direction in fish monitoring. *PLoS ONE*, 4(10):e7351, 10 2009.