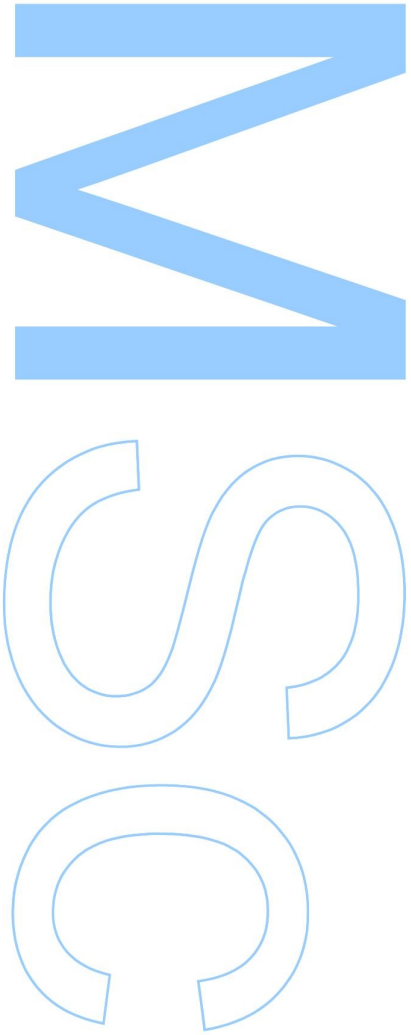
Dolphin: A Domain-Specific Language for Autonomous Vehicle Networks



Keila Mascarenhas de Oliveira Lima
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2017

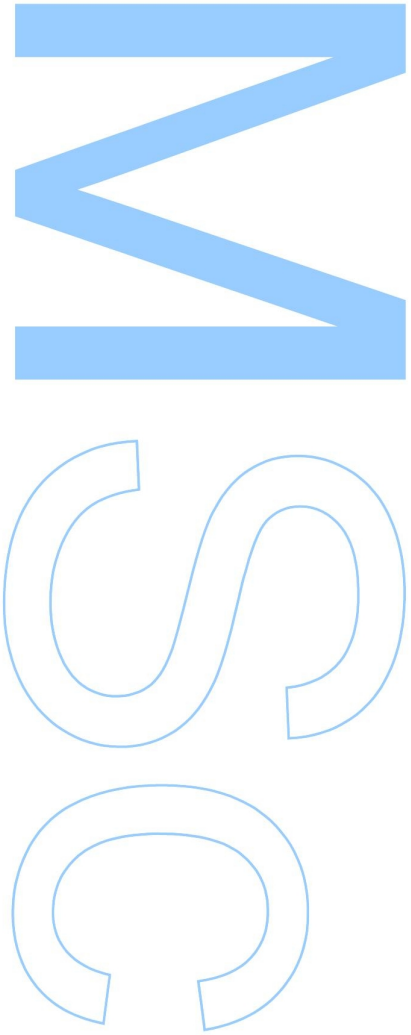
Orientador
Eduardo Resende Brandão Marques,Professor Auxiliar Convidado, Faculdade de Ciências da Universidade do Porto

Coorientador
José Carlos de Queirós Pinto, Assistente de Investigação, Faculdade de Engenharia da Universidade do Porto

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

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

I would like to thank my supervisors Eduardo and José for all the support and guidance through this jorney. It has been a year full of knowledge and learnings.
I would also like to thank my colleagues at LSTS and professor João Sousa for the collaboration and availability shown during this period.
All this process could not be possible without the inconditional love and support of my family and friends that helped me keep the balance between my academic and personal life.
At last I would also like to thank the Capeverdian government for the financial support through my academic journey in Portugal and the oportunity to study abroard.

Agradecimentos

Gostaria de agradecer os meus orientadores Eduardo e José por todo o apoio prestado durante esta jornada. Tem sindo um ano cheio de conhecimento e aprendizagem.
Gostaria também de agradecer aos meus colegas do LSTS e ao professor João Sousa pela colaboração e disponibilidade demonstrada durante este período.
Todo este processo não seria possível sem o carinho e apoio incondicional da minha família e amigos, que ajudaram a manter o equilíbrio entre a minha vida académica e pessoal.
Por fim, gostaria de agradecer ao governo Caboverdiano pelo apoio financeiro durante o meu percurso académico aqui em Portual, e pela oportunidade de estudar no estrangeiro.

Abstract

Current research efforts regarding autonomous vehicles lead to a intensification of its usage in the real world as applied to several operational scenarios. An example of such is the usage of sets of autonomous underwater, surface and aerial vehicles in oceanographic and military operations, working together to achieve a common goal. These vehicles are composed of sensors and actuators that allow them to complete their tasks. The usage of its capabilities is only possible with the support systems, software and human operators that interact with these vehicles forming a heterogeneous network. Apart from these characteristics, we have to take into account that the environment in which these vehicles operate does not favour the complexity of its operation, increasing the necessity to develop tools to help simplify the interaction within these networks.

With this motivation, we present a domain-specific language called Dolphin for coordinated task execution in a network of autonomous vehicles. Dolphin expresses tasks that are allocated to vehicles dynamically selected from a network, in integration with an open-source toolchain developed at the Laboratório de Sistemas e Tecnologia Subaquática (LSTS)/FEUP. Dolphin tasks are defined compositionally for multiple vehicles through operators for concurrency, sequence or event flow, building on the base case of elementary tasks expressed as maneuver plans, encoded in the IMC protocol developed by LSTS.

Dolphin is implemented in Groovy/Java and has been integrated with Neptus, the command and control software also developed by LSTS, and has been evaluated in several field tests, including open-sea tests conducted in collaboration with the Portuguese Navy.

Resumo

A investigação que tem sido feita na área de veículos autónomos resultou na intensificação da utilização dos mesmos em casos reais aplicados a vários cenários de operação. Como exemplo dessa utilização temos o uso de conjuntos de veículos autónomos subaquáticos, de superfície e aéreos em operações oceonográficas e militares, trabalhando em conjunto para atingir um objetivo comum. Estes veículos são compostos por sensores e actuadores que os permitem completar as suas tarefas. A utilização das suas capacidades é possível devido a sistemas de suporte às operações, *software* e operadores humanos que interagem com os veículos formando uma rede heterogénia. Para além destas características, temos que ter em conta que o ambiente em que os veículos operam fazem aumentar a complexidade das operações, fazendo aumentar a necessidade do desenvolvimento de ferramentas que simplifique a interação nessas redes.

Com essa motivação, apresentamos a linguagem de domínio específico, denominada Dolphin, para coordenação da execução de tarefas em redes de veículos autónomos. A Dolphin expressa tarefas que são alocadas a veículos selecionados dinamicamente na rede, em integração com as ferramentas de *software* de código aberto desenvolvidas no Laboratório de Sistemas e Tecnologia Subaquática (LSTS)/FEUP. Essas tarefas são definidas de forma composicional para múltiplos veículos através de operadores de concorrência, sequência ou de fluxo de eventos, construíndos sobre tarefas elementares expressas como planos de manobras, codificadas no protocolo IMC desenvolvido pelo LSTS.
A linguagem Dolphin foi implementada em Groovy/Java e foi integrada com Neptus, o *software* de comando e controlo também desenvolvido pelo LSTS, tendo sido avaliada em vários testes de campo, inclusive em testes em mar aberto conduzidos em colaboração com a Marinha Portuguesa.

Contents

List of Figures

List of Tables

Acronyms

ACCU Android Command and Control Unit

APDL Administração dos Portos do Douro, Leixões e Viana do Castelo

API Application Programming Interface

ASV Autonomous Surface Vehicle

AUV Autonomous Underwater Vehicle

CCU Command and Control Unit

COTS Commercial Off-the-Shelf

DRiP Douro River Plume Tracking

DSL Domain Specific Language

GUI Graphical User Interface

IDE Integrated Development Environment

IMC Inter-Module Communication Protocol

LAUV Light Autonomous Underwater Vehicle

LSTS Laboratório de Sistemas e Tecnologia Subaquática

NVL Networked Vehicle Language

REP Rapid Environment Picture

ROV Remotely Operated Vehicle

UAV Unmanned Aerial Vehicle

UUV Unmanned Underwater Vehicle

Chapter 1

Introduction

1.1 Motivation

The research and investment on autonomous vehicles has been gradually increasing lately due to their application in multiple scenarios such as scientific surveys, military and transportation [9]. Its usage is also less expensive and easy to handle than manned or remotely-operated vehicles as they do not require human operators to be attending their (often long and dull) operations. One aspect that increases the number of possible scenarios is to coordinate the execution of multiple autonomous vehicles, possibly of different types (aerial, surface and underwater) with configurable payloads, changing the requirements in each mission which increases the number of resources that the operators have to manage, making their task more difficult. Another characteristic to take into account is the physical mean where these vehicles are deployed which has limited communication resources as the acoustic communication used to communicate with the vehicles when they are submerged.

These vehicles are used in cooperation, combining their capabilities to execute tasks in order to accomplish a common goal. Therefore it is essential the development of tools to simplify multiple vehicles monitoring and operation taking into account the environment and conditions which they operate. In the last few years, the Laboratório de Sistemas e Tecnologias Subaquáticas (LSTS) has been working in the development and deployment of autonomous vehicles having a toolchain of open-source software to support their operations. In the recent work developed in LSTS, we can find the Networked Vehicles' Language (NVL [15, 14, 28]) and multiple vehicle

planner [27] which demonstrate some of the efforts and interest in the coordination of multiple autonomous vehicles.

1.2 Problem Statement

This thesis addresses the specification of system-level multiple vehicle behaviour and its execution by heterogeneous vehicles in communications-restricted environments. Even though command and control software generally allows the manipulation of vehicles individually, the increase in the number of vehicles in real-world applications makes joint coordination hard to program and automate. One of the main advantages of multiple heterogeneous vehicles comes from the possible combination of their different capabilities filling possible gaps that might appear in their individual command. So there is a need to coordinate the integrated use of these vehicles facilitating the operators load.
We developed Dolphin in an attempt to address this problem. Dolphin follows on a prior effort with similar goals, the NVL language [15, 14, 28]. In Dolphin we strived for more expressiveness and flexibility, easier programming, and a tighter integration with the LSTS toolchain programming. Conceptually and technically, Dolphin builds on the valuable experience acquired with NVL, but, considering the goals above, a decision was made to design and implement Dolphin from scratch.
Validation of the language in real operation scenario is required in order to adjust the behaviour to mutable conditions. To do so, LSTS's software toolchain was integrated with the language, making possible its usage with their systems.

1.3 Contributions

The contribution of this thesis comprises three main aspects:

1. The Dolphin language itself, expressing tasks that are allocated to vehicles dynamically selected from a network. Dolphin tasks are defined compositionally for multiple vehicles through operators for concurrency, sequence or event flow, building on the base case of elementary tasks provided by a base platform. The core of the language is implemented

in Groovy and Java, facilitating integration with target platforms, and has an extensible design.

2. Dolphin has been integrated with the open-source LSTS toolchain. The base case of elementary tasks is expressed in the LSTS toolchain as maneuver plans, encoded in the IMC protocol. This takes form as a plug-in for the Neptus command and control software, and a stand-alone runtime that builds solely on IMC. Furthermore, a IMC DSL and a Neptus Groovy plug-in were also developed, aiding in the definition of IMC plans.

3. Dolphin has been evaluated in several field tests, including open-sea tests in collaboration with the Portuguese Navy, demonstrating its applicability to complex real-world scenarios.

1.4 Thesis Structure

The rest of this thesis is organized as described below:

- Chapter 2 presents the background for this thesis in terms of work at LSTS, operational scenarios in autonomous vehicle networks, and survey on related work.

- Chapter 3 presents the Dolphin language in terms of its architecture and main features, along with the related developments of the IMC DSL and the Neptus Groovy plug-in.

- Chapter 4 describes the design and implementation of Dolphin and the other components mentioned above.

- Chapter 5 presents results from field tests used to validate our approach.

- Chapter 6 ends with a discussion of accomplishments, limitations and directions for future work.

- Some appendices (A to D) are provided to support the contents presented in the evaluation chapter, such as scripts tested during the operations, logs fragments and execution timelines.

Chapter 2

Background

In this chapter we present the necessary material for a better understanding of the subject issued in this thesis. We introduce the Laboratório de Sistemas e Tecnologia Subaquática in Section 2.1, describing the currently used systems and most relevant software of the laboratory toolchain. We continue with the description of some operating scenarios for autonomous vehicles in Section 2.2. Afterwards, in Section 2.3, we present the NVL language that demonstrates the previous efforts made to issue the problem addressed in this thesis. In Section 2.4 we present some implementations used to model robots operations which includes some DSLs. We finished with a brief description of the Groovy language in Section 2.5, in which Dolphin DSL was constructed.

2.1 Laboratório de Sistemas e Tecnologia Subaquática

LSTS has been working in the design and construction of unmanned ocean and air vehicles along with support systems and software for their operations, including not only the software and communication protocols used by the unmanned vehicles but also console software that allows human operators to supervise and interact with such systems.
In this section we review some of the work developed in LSTS relevant to this thesis which includes the currently used vehicles, the manta communication support system and the main components of the LSTS software toolchain (Neptus-IMC-Dune).

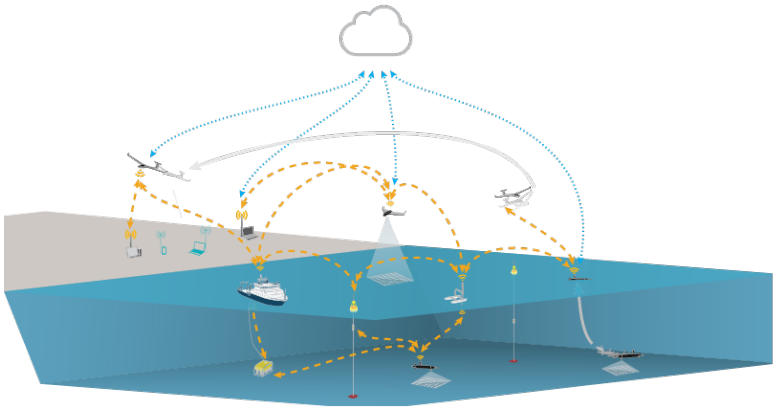


Figure 2.1: LSTS systems scenario.

2.1.1 Autonomous Vehicles

In this section we describe some of the systems currently used at LSTS for a better understanding of the missions and operational scenarios described at the section 2.2 and in the results at the chapter 5. To do so we presented a table with some of the most relevant specifications of the vehicles, explaining their operation mode. Notice that LSTS has developed other systems but they are not part in the actual operational scenario. All the vehicles are made with modular components which facilitates their construction, maintenance and operation [9].

LAUV

These man-portable vehicles are the most used systems at LSTS, which can endure up to 24 hours while travelling at 3 knots. In the table 2.1 we presented some characteristics these LAUVs. All of them have support for wireless communication via WIFI, GSM and acoustic modulation between compatible modems installed either in other vehicles or in the Manta gateway (Section 2.1.1). In addition, some LAUVs have support for communication via satellite with an Iridium antenna as we can see in the hardware configuration column on the table 2.1 bellow. These vehicles can be configured with additional sensors, considered as payload.
In terms of navigation, LAUV vehicles can use GPS whenever they are at the surface but need to use alternative sensors when they are underwater. Bottom-mapping LAUVs usually use an Inertial Navigation System (INS)

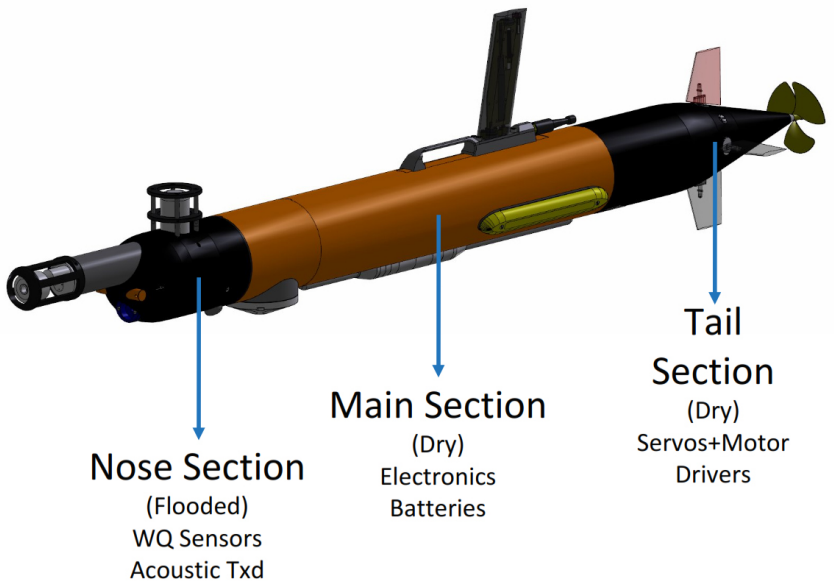


Figure 2.2: LAUV model.

composed by a compass, accelerometers and gyroscope as well as a Doppler Velocity Log (DVL) acoustic sensor that measures the vehicle velocity in relation to the bottom. Oceanographic LAUVs usually do not have DVL and use a much cheaper MEMS-based INS system as they do not require high navigation accuracy. Another additional configuration common to all LAUVs is a secondary CPU that, in the case of the *lauv-noptilus-3*, is used for the camera. Environmental payload to measure turbidity, pH, crude oil or chlorophyll can also be configured (one at the time) in the front of the nose section of the vehicles (Figure 2.2).

UAV

LSTS has being using several different UAV model types along the years. In recent deployments, LSTS UAV fleet has been composed mainly by two UAV types: X8 which is a fixed wing UAV (plane Figure 2.3) and the Mariner (copter-style Figure 2.4). These UAVs have a shorter autonomy ranging from 20 minutes of operation up to 50 minutes (table 2.2), having low production costs since they use several commercially available off-the-shelf (COTS)

Table 2.1: LSTS LAUV Specification.

Vehicle	**Length (cm)**	**Width (cm)**	**Max Depth (m)**	**Hardware Configuration**
lauv-arpao	137	30	100	Iridium
lauv-noptilus-1	197	30	50	CTD, DVL, IMU, Multibeam, Sidescan
lauv-noptilus-2	193	30	50	DVL, Echo Sounder, IMU, Sidescan, Sound Velocity Sensor
lauv-noptilus-3	185	30	50	Camera, DVL, Multibeam, Echo Sounder, IMU, Sidescan, Sound Velocity Sensor
lauv-xplore-1	184	30	60	CTD, Iridium, WET Probes
lauv-xplore-2	200	30	60	Iridium, WET Probes
lauv-xtreme-2	205	30	100	DVL, Iridium, Sidescan, Sound Velocity Sensor

components such as the frame and the autopilot (Pixhawk). Their payload includes high-resolution recording and streaming cameras used for mapping or surveillance and with Wi-Fi range up to 30 kms which allows the usage of this vehicles as a network extension (data mules).

Figure 2.3: UAV X8.

Figure 2.4: UAV Mariner.

Table 2.2: LSTS UAV Specification

Vehicle	**Length (cm)**	**Width (cm)**	**Endurance**	**Max Altitude (m)**
Mariner	55	55	20 min	1000
X8	60	212	50 min at 17 m/s	600

Manta Communications Gateway

The Manta (Figure 2.5) is a communication gateway that allows communication between software modules at the operational station and the vehicles. The communication is made via WIFI/3G/Iridium or acoustic. It allows communication between different means such as underwater, surface and air, forwarding messages to the nodes connected to them. As an example, when the vehicles are submerged, they can send data via an acoustic modem which is received in a similar one connected to the Manta that converts the received data to UDP datagrams to be disseminated in the network. This happens when a vehicle sends an acoustic report which includes minimal information, having the Manta to convert it to an IMC message, filling some parameters, so it can now be understood by the other nodes.

Figure 2.5: Manta Communications Gateway.

2.1.2 Software

As we mentioned before, LSTS has been developing open-source software[1] [25] that supports the vehicles operation. We describe bellow some of the LSTS software toolchain modules that somehow interacts with the implementation made for this thesis: the IMC communication protocol, the Neptus command and control software and the software running onboard of the vehicles, DUNE.

IMC

The IMC (Inter-Module Communication [16]) is the communication protocol for vehicles and sensors networks based on messages. It is used in all the systems at LSTS ensuring interoperability between heterogeneous components, such as the vehicle-to-vehicle communication or even internally between different components from the vehicle. The protocol message set is defined in XML and it can be parsed in order to generate bindings for Java, C++ and Python.

There are different types of messages (Figure 2.6), matching the levels of control and capabilities of the vehicles:

- **Mission Control Messages** that allows the definition of a mission (graph of maneuvers) life cycle.
- **Vehicle's Control Messages** allows the interaction with the vehicle,

[1] LSTS software toolchain is available in `http://github.com/LSTS`

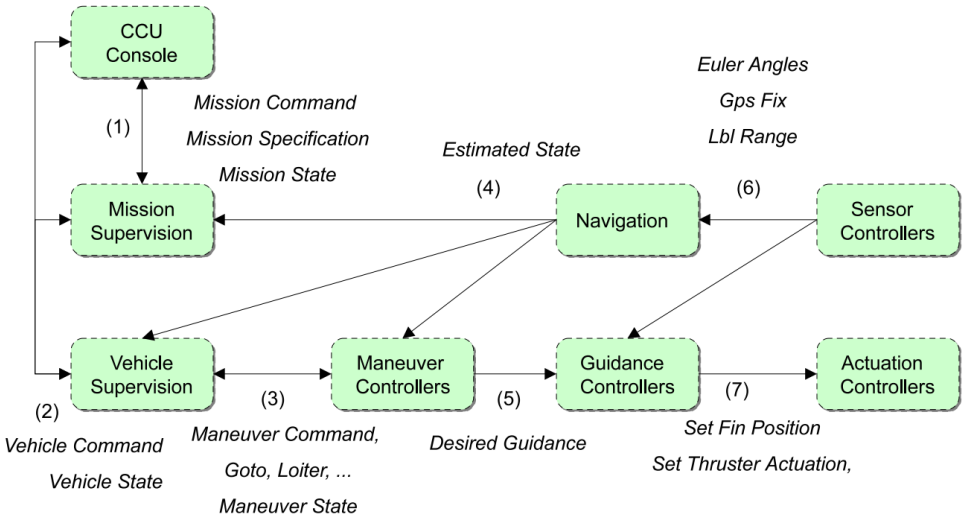


Figure 2.6: IMC messages flow from IMC.

typically used in external sources or supervisor modules. It can be used to command or monitor vehicle's execution. An example of the subscription to this type of messages can be found in the Figure 2.1.2,

- **Maneuver Messages** are used to define maneuvers and also actions and states associated to their execution. As an example, in the Figure 2.1.2, we present an extract of Java code to create a plan start request with two Goto maneuvers. We can notice that both maneuvers, which are an IMC message type, are embedded in a plan specification message that is used as an argument in the PlanControl type exemplifying the message inlining in IMC.
- **Guidance Messages** defines the vehicles parameters such heading, depth and velocity in each step.
- **Navigational Messages** reports the vehicle's relative navigational state.
- **Sensor Messages** reports the data collected by the sensors.
- **Actuators Messages** defines the interaction with the actuators hardware controller which can be used in the vehicle's guidance.

```
1  public class MessageListening {
2
3      @Consume
4      public void onState(EstimatedState state) {
5
6          System.out.println("Received Estimated State from "+
7              state.getSourceName()+":");
8
9          System.out.println(state.getX()+", "+
10             state.getY()+", "+state.getZ());
11     }
12
13     @Consume
14     public void onAnnounce(Announce ann) {
15         System.out.println("Received Announce from "+
16             ann.getSourceName());
17     }
18
19     public static void main(String[] args) {
20         IMCProtocol protocol = new IMCProtocol(6006);
21         protocol.register(new MessageListening());
22         try {
23                 Thread.sleep(100000);
24         }
25         catch (Exception e) {
26                 e.printStackTrace();
27         }
28     }
29 }
```

Figure 2.7: Example of Message Listening in Java implementation of IMC from `github.com/LSTS/imcjava/`

DUNE

The DUNE (DUNE Uniform Navigational Environment [25]) is the onboard software used in the vehicles, Manta and oceanographic buoy. The software is written in C++, requiring only the standard library of the language which allows the portability to different operating systems (Linux, Windows, SunOS) and CPU architectures (ex. x86, ARM,Sum SPARC) to real-time

```
// Create goto maneuver
Goto gt = new Goto()
                .setLat(Math.toRadians(41))
                .setLon(Math.toRadians(-8))
                .setZ(2)
                .setZUnits(ZUnits.DEPTH)
                .setSpeed(1000)
                .setSpeedUnits(SpeedUnits.RPM);

// Create another goto maneuver based on previous one
Goto gt2 = new Goto(gt)
                .setLat(Math.toRadians(41.0001));

// Create a plan start request with the 2 maneuvers
PlanControl cmd = new PlanControl();
cmd.setArg(PlanUtilities.createPlan(planId, gt, gt2));
cmd.setOp(OP.START);
cmd.setRequestId(1);
cmd.setType(TYPE.REQUEST);
```

Figure 2.8: Example of Plan start request in Java implementation of IMC from `github.com/LSTS/imcjava/`

tasks programming. DUNE is divided in tasks that represents different modules according to the associated layer of control, sensor, actuator, monitors, supervisors or communication interfaces. Each DUNE task is executed concurrently and in a modular way communicating with each other using the publish/subscriber pattern [24]. The tasks exchange information with each other using IMC through a message bus which is illustrated in the Figure 2.9. DUNE also has support for a simulation mode allowing the users to test their experiments or reproduce an mission prior to the deployment on real autonomous vehicles.

Neptus

Neptus is the command and control open-source software developed in LSTS that is used in multiple phases of the autonomous vehicles operation as [25]:

- **Mission Planning** (console in Figure 2.10) phase can be done before the actual mission, where the operators have the ability to configure or

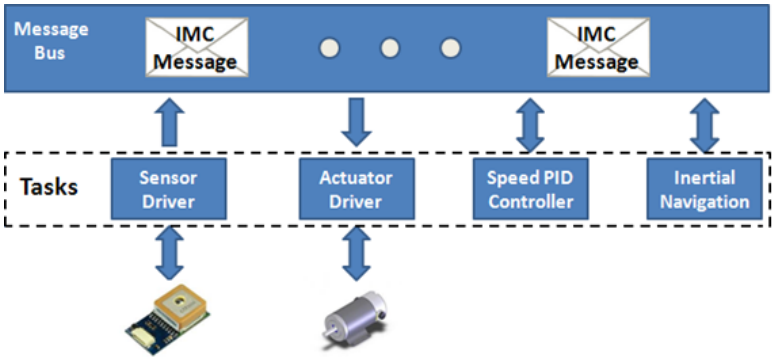


Figure 2.9: DUNE: tasks and message bus from [25].

create a console defining the operational area in the map and projecting the plans that can be saved on the console for future use. The plans specification is done visually allowing the users to edit each maneuver and its parameters along with payload definition according to the target vehicles. Besides the map and plan configurations, the users can also configure some vehicles parameters such as the frequency of the acoustic reports or operational limits.

- **Execution and Monitoring** is made in real-time during the missions allowing users to intervene in order to start, stop/abort or adjust the plans being executed even while the vehicles are underwater, by using acoustic communications.

- **Mission Review and Analysis** (Neptus MRA console in Figure 2.11) happens after the missions, being used to visualize the data collected or generated during vehicles operation. This feature gathers and presents vehicles logged data, having also support for mission replay. The information is presented in different customizable plots and reports being able merge them to other vehicles logs and even export them to different formats.

The software is written in Java which allows its portability to different operating systems, having an easy to extend architecture through plug-ins development. The plug-ins are dynamically included in the software libraries [24], allowing the addiction of new features both in the planning/ex-

ecution console and in the mission review and analysis console. These new features can be selected configuring the consoles, allowing the users to shape them according to their needs. As the other systems in LSTS, Neptus also uses IMC to communicate with the other nodes in the network having a manager to mediate the protocol usage.

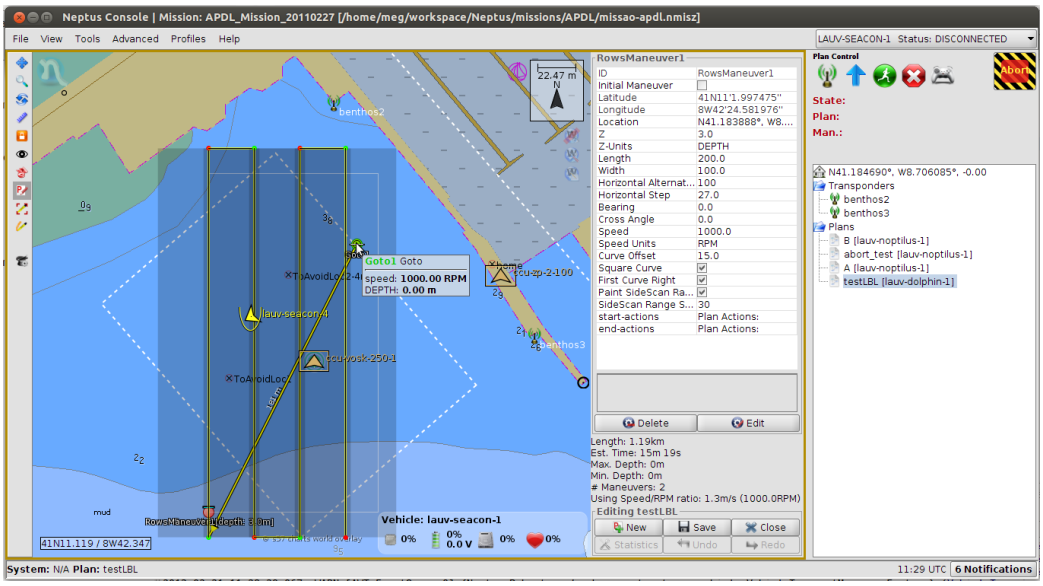


Figure 2.10: Neptus - Command, control and vehicles monitoring console.

2.2 Operational Scenarios

In this section we describe some operational scenarios, some of them contemplated by LSTS, which these autonomous vehicles are used to a better understating of some examples in the upcoming sections/chapters[2].

2.2.1 Tracking

In tracking we can identify two scenarios, both involving AUVs: one of them is made iteratively according to the water parameters collected and the other

[2]Concrete mission descriptions of LSTS operation scenarios are also presented in Chapter 5.

Figure 2.11: Neptus - data visualization and analysis.

is more spaced in time since the vehicles movement dependent on volatile entities.

Plume Tracking

This scenario requires only AUVs which perform samples searching for environmental disturbance (coastal fronts, pollution, etc). Multiple AUVs can be deployed, in a pre-defined region, to map this disturbance moving iteratively according to a certain criteria that adjusts their movement. Each vehicle has a region to map per iteration making environmental measurements and following a defined gradient to determine local maximum and minimum. To make these measurements, the vehicles are equipped with temperature [3], redox, salinity, chlorophyll and/or turbidity sensors attempting to infer the pollution source (or any other distinct water property).
Currently, LSTS has been doing missions to track the Douro river plume and detect their front involving two AUVs equipped with environmental sensors.

Fish Tracking

In this case the objective is to track one or more marine species identifying them in different ways. In 2014 LSTS tested the detection and tracking of sunfishes marked with tags that sent the current location by satellite whenever they were at surface [26]. After the fishes are detected, both AUVs and UAVs were sent to the location to survey the environment surrounding them. An alternative way to detect the targets evolve the usage of acoustic transducers in the tags along with one or more AUVs, measuring the distance to the target from different positions. This method would allow the triangulation of the transducers position [19]. After detecting the position, the vehicle(s) follow(s) it, switching between moving in their direction and detecting a new transducer position.

2.2.2 Data Muling

A data muling scenario appears with the necessity to transfer data between different locations (normally remote ones). A possible scenario is the transmission of data to operators located in a base station [7] for processing and interpretation, required to defined the next step or task in a mission. This scenario involves vehicles with two distinct roles, being one group responsible for the data collection, and the other responsible for transmission of the collected data.
For example, in the sunfish tracking scenario previously described [26], UAVs were also used as "data mules". Another example, using UAVs, is referred in [14] and described in section 2.3, being revised in chapter 5, where we recreated this scenario in two similar approaches with Dolphin scripts. In this scenario the UAV performs a rendezvous sequentially with the AUVs, collecting the data from them.

2.2.3 Mapping

In a mapping scenario an area (typically polygonal) of interest is mapped autonomously by vehicles normally equipped with proper sensors to generate images when their data are processed. An example of mapping using sonar sensors can be found in Figure 2.12, where an AUV measures the sea bed bathymetry using a multibeam sonar and the geomorphology of the bottom using a sidescan sonar. This type of sensors can scan laterally the bottom of

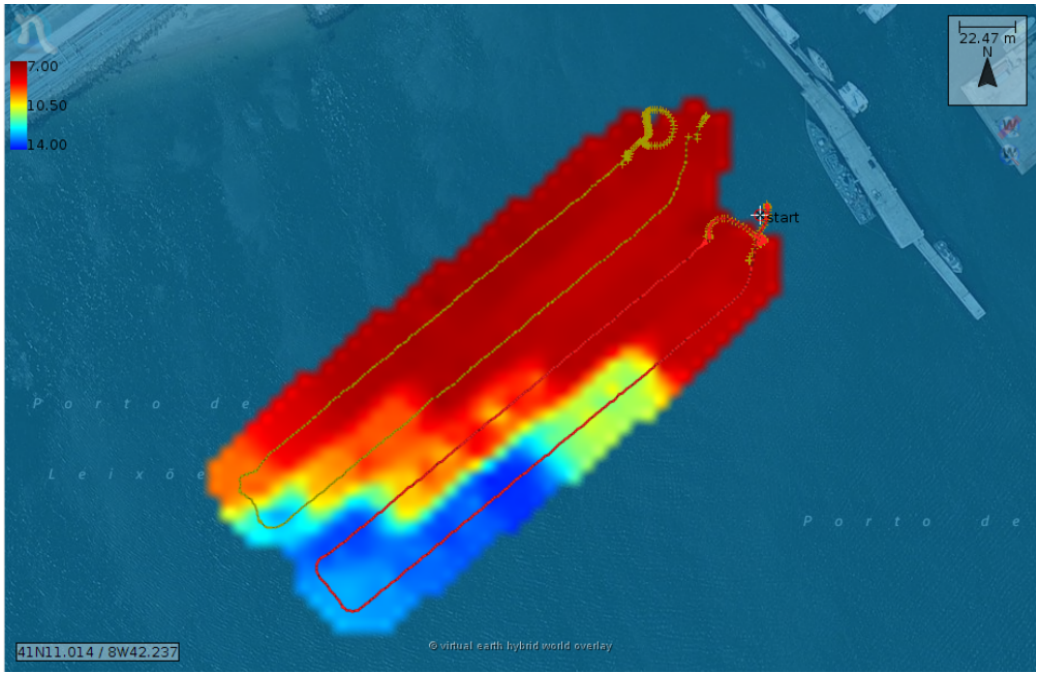


Figure 2.12: Bathymetry data rendered in Neptus MRA from [25]

the sea/river around 50 meters of distance from the vehicles, being necessary to move them in order to map the whole area of interest. To do so, this area can be divided to multiple vehicles so that their trajectory covers totally the area of interest.
UAVs can also be deployed in mapping scenarios [18], being able to cover a larger area using photogrammetry. These vehicles are equipped with payload to allow image acquisition and geo-referencing used for 3D reconstruction based on the image orientation and the camera calibration parameters.

2.2.4 Patrolling

In patrolling scenarios, an area of interest is also defined (e.g. a border), requiring periodic coverage normally performed by more than one autonomous vehicles. If the surveillance is continuous, it has to support vehicle replacement in order to recharge or in case of error.
As an example, the Portuguese coast surveillance is made by manned fighter jets that could be replaced by multiple UAVs in the future that could lead

to a drastic decreasement of costs.

2.3 NVL Language

The Networked Vehicles Language was developed to coordinate multiple vehicles, having their initial prototype been described in [15, 14, 28]. In this section we review the language architecture, implementation and expressiveness discussing some limitations and improvements that can be done, some of them addressed in this thesis.

2.3.1 Expressiveness

In a NVL program we are able to globally coordinate vehicles selected in the network, allocating tasks to be executed by them. The base primitive of the language allows the allocation of one task to one or more vehicles with an associated time interval. It has other primitives which gives support to sequential, concurrent, control flow and time restrictions in a program. To ilustrate NVL's expressiveness we can look into an example on Figure 2.13 from [14]. It addresses a data mulling scenario where there is data collection by AUVs[3] followed by a sequential rendezvous between an UAV and each AUV. We can notice the main procedure where the program execution starts having some auxiliary procedures (rendezvous in line 32) and declarations (tasks definitions). The spacial distribution of the tasks and the execution flow are shown in the Figures 2.14 and 2.15 respectively.
The program in Figure 2.13 begins with tasks declarations which specification were defined outside the program, defining three survey areas (area_1, area_2 and area_3) and a cooperative rendezvous task (RV, lines 5-6). The vehicles selection is made inside the main procedure with the **select** instruction (lines 10-11) with an associated time limit of 5 minutes. The instruction step (line 14), inside the **then** block makes the execution of the surveys concurrently also with a time constraint, this time for 60 minutes. Concluded the surveys tasks, the UAV is selected with 5 minutes time limit (line 20) to perform the three rendezvous tasks sequentially (lines 24-26). The rendezvous is defined in a procedure (lines 32-38) where an UAV collects data from one AUV at the time in 15 minutes cycles having the step primitive to activate the RV task.

[3] Also denoted as UUV in the program in Figure 2.13.

```
// Task declarations
task area_1 (vehicle uuv);
task area_2 (vehicle uuv);
task area_3 (vehicle uuv);
task RV (vehicle uav, vehicle uuv)
  yields done, moreData;
// Main procedure
proc main() {
  // Select UUVs.
  select 5 m {
    uuv1 uuv2 uuv3 (type: "UUV")
  } then {
    // Execute sampling tasks.
    step 60 m {
      area_1(uuv1)
      area_2(uuv2)
      area_3(uuv3)
    }
    // Select UAV.
    select 5 m {
      uav (type: "UAV")
    } then {
      // Execute rendezvous tasks.
      call rendezvous(uav, uuv1)
      call rendezvous(uav, uuv2)
      call rendezvous(uav, uuv3)
      message "done"
    }
  }
}
// Rendezvous execution
proc rendezvous(vehicle uav, vehicle uuv) {
  do {
    step 15 m {
      rvRes = RV(uav, uuv)
    }
  }
  while (rvRes = moreData)
}
```

Figure 2.13: Programa NVL retirado de [14]

In the Figure 2.16 we can see a possible execution timeline of the program described above.We can see that the surveys tasks have to be done in one hour and the rendezvous is performed after that time even if they finish early. We can also note that the rendezvous are performed in a strict order: uuv1, folowed by uuv2 and then uuv3.

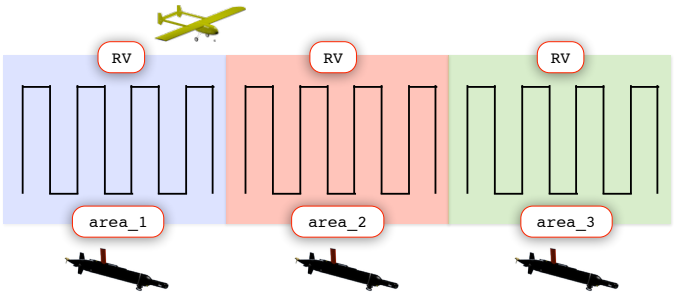


Figure 2.14: NVL program example - spacial distribution.

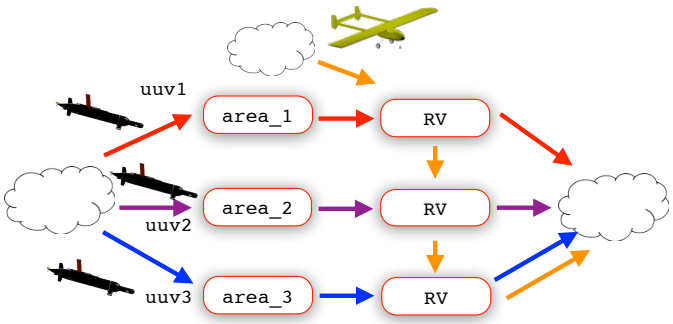


Figure 2.15: NVL program example - task flow.

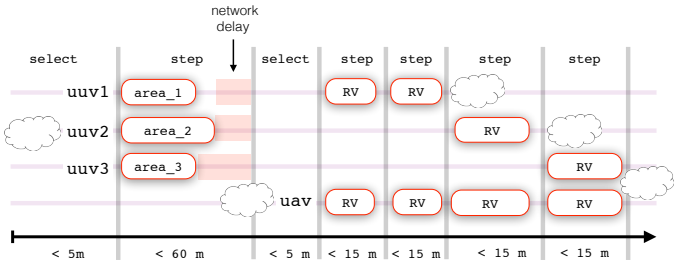


Figure 2.16: NVL program execution.

2.3.2 Architecture

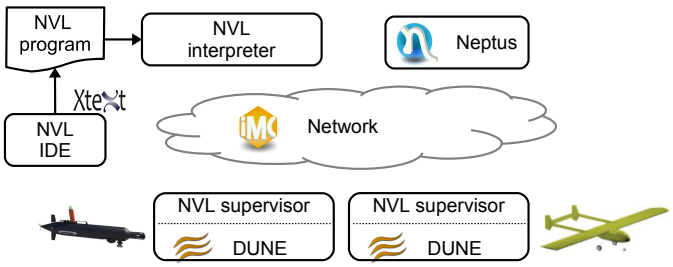


Figure 2.17: NVL Architecture.

The language prototype architecture is illustrated in the Figure 2.17 which is formed by the following components:

- NVL editor where a NVL program can be edited in the Eclipse IDE using a plug-in that implemented the Xtext framework for DSLs. This plug-in allowed the edition and validation of programs in a user friendly way.
- NVL Interpreter which executes a validated NVL program and communicates with supervisors modules running onboard of the vehicles.
- NVL Supervisor that mediates the access to the vehicles, answering to requests from the interpreter and interacting with DUNE instances to make these requested tasks be accomplished locally.
- Neptus can be used by operators to define the tasks as IMC plan specifications that can also be sent to vehicles in the console.

2.4 Related Work

Contrasting general purpose language, DSLs are implemented to model a specific problem/domain. The level of abstraction of a DSL should be as the same level as the level of abstraction in the problem [11]. Domain Specific Languages (DSL) can be qualified as internal or external, taken into account the way they are implemented. External DSLs are almost seen as new languages since they require the implementation of processes like parsers and runtime/interpreter. The semantics and syntax depends on the level of complexity deployed in the language implementation. In the other hand, internal DSLs (also known as embedded DSLs) take advantage of existing languages to implement their semantics. The syntax can be limited to the host language available infrastructure. Java Virtual Machine (JVM) languages presents features that make them good candidates to host internal DSLs (described in [11]), besides the portability to different architecture offered by the platform.
We present in this section some DSLs for robots, focusing on the ones that were designed or tested with autonomous vehicles being aerial, surface, underwater or even ground (see Section 2.4.2).

2.4.1 Common Control Language

Common Control Language (CCL) is a language for distributed problem solving and can be deployed to control multiple autonomous unmanned vehicles [8]. The initial objectives of the language were to standardize the communication between the different agents and support the interaction between operators and machine [6]. The language is composed by an onboard interpreter that receives tasks defined in CCL and translates them to processes directives. These directives are instantiated with the Distributed Control Environment (DICE) which was also implemented in the vehicles to maintain the processes. CCL messages are reduced to a few bytes due to acoustics communications typical in AUVs operations in which the language was tested.

2.4.2 Buzz

Buzz is a programming language for robot swarms [23]. It has support for single-robot with data exchange between neighbours and swarm-based prim-

itives with global information sharing within a swarm. Buzz Virtual Machine (BVM) was written in C language and the language runtime platform can be placed on top of Robot Operating Systems frameworks, being easily extended with new primitives according to the underlying system capabilities/features. In Figure 2.18 we presented a buzz program adapted from [23]. There are three different swarms: ground one grouping by wheels symbol (line 8), another grouped by grip symbol (line 11) and the last one resulting from the intersection of the two (line 13). In the program there is also task definition and execution (lines 5 and 14 respectively).

```
1  # Group identifiers
2  GROUND = 2
3  GRIPPERS = 4
4
5  # Task for ground-based gripper robots
6  function ground_gripper_task() { ... }
7
8  # Create swarm of robots with 'set_wheels' symbol
9  ground = swarm.create(GROUND)
10 ground.select(set_wheels)
11
12 # Create swarm of robots with 'grip' symbol
13 grippers = swarm.create(GRIPPERS)
14 grippers.select(grip)
15
16 # Assign task to ground-based gripper robots
17 ground_grippers = swarm.intersection(GROUND + GRIPPERS, ground,grippers)
18 ground_grippers.exec(ground_gripper_task)
```

Figure 2.18: Buzz example program adapted from [23]

2.4.3 BigActors

BigActors is a hybrid model that combines the actors models and bi-graphs to model structure-aware computations in time and space [20]. This model is used to manage and control heterogeneous mobile robot systems, being the actors the computing agents capable of sending asynchronous messages. These actors are embedded in a spacial environment modelled as bi-graphs

that specifies machines locations and connectivity. BigActors model is implemented in BigActors programming language which is a DSL embedded in Scala. The DSL is an extension of the Scala Actor Library with BigActors commands and implicit conversions to support the syntax [21]. The language provides the means for actors to migrate, control and observe over the world structure modelled in bi-graph. BigActors programs interacts with a Logical-Space Execution Engine (LSEE) which executes upon Robot Operating Systems middleware allowing the interaction with the physical space [22].

2.4.4 Planning Domain Definition Language

Planning Domain Definition Language(PDDL [17]) is general language to specify domains and planning problems. The language syntax includes functions, predicates, actions and objectives that can be used to define effects, objects, pre and post conditions. PDDL has been extended to support temporal and numerical properties of the domains for planning [10]. The fact that it combines problems descriptions with domains allows planning independence being able to specify different problems for the same domain definition, each one corresponding to different planning problem.
In recent work developed in LSTS [4], PDDL was used to express planning problems presented in the example in Figure 2.19. Tasks specifications defined by operators through Neptus and they were automatically translated to PDDL. A planning module generated plans that were sent to vehicles in case the restrictions specified in PDDL were feasible.

2.4.5 Comparison

We can divide the presented DSLs in groups according to their level of expressiveness and propose. We have low level and concise languages which the main concern are the communication means characteristics (error prone and low bandwidth), generally acoustic. Theses languages are normally used with underwater and surface vehicles functioning almost as communication protocols such as the CCL DSL.
In the other hand we have more flexible languages with concerns at level of coordination and planning of the vehicles operations using communication protocols libraries associated to the vehicles systems. Buzz, BigActors Scala DSL, NVL and PDDL can fit into this category, besides the differences in the level of abstraction.

```
(: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 2.19: PDDL specification from [4].

2.5 Groovy

Groovy was the main tool used in the implementation of Dolphin. Therefore we describe the language, highlighting some of its features [5] that were somehow used in the implementation or relevant for understanding this programming language. It is a dynamic language for the Java platform, being part of the range of languages that use the Java Virtual Machine (JVM) which works on different hardware architectures providing portability to these languages.

2.5.1 Features

- Closures - in Groovy a Closure is a fragment of code whose variables can or cannot be bind, represented between braces and they can be assigned to a variable. It origins from lambda calculus theory which is the foundations for functional programming.
- Operators overloading - there are a range of operators that can be implemented defining behaviour
- Optional typing - we have the choose to declare variable types or let the work to be done when the code is evaluated, in the last case making

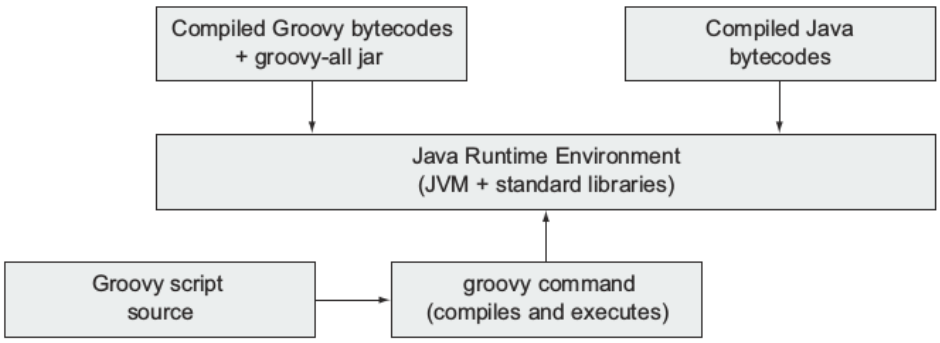


Figure 2.20: Groovy architecture in the Java platform from [13]

the code less verbose.

- Flexible syntax - this feature allows programmers to write code seen as "syntactic sugar" to strict Java syntax making it easier to understand. An example of this can be found in function calls where there is no need to add parenthesis if there is at least one argument. Moreover, Java code syntax is also well-formed Groovy code.

- MetaClass - this is one of Groovy metaprogramming features which allows the addition of methods and properties to native types as Number or Strings, or newly defined types and also allows adding behaviour to classes defined elsewhere easily.

These are some of the language characteristics that can be customized, creating intuitive programs which makes the language suitable for implementing DSLs.
An example of a Groovy DLS is the Gradle build system, which is written in Groovy and Java due to the project dimensions [13]. It can be a shorter and easier way to compile projects, allowing definition of tasks and parameters dependencies, repositories or versions compatibility . In Figure 2.21 we present a basic Gradle script where we can notice Groovy capabilities for scripting. In the script, source and test directory are defined (lines 7-9 and lines 10-13 respectively), specifying a different project structure (not strict as Maven). It is also defined dependencies including all *jar* files from a specific directory(lines 2-4).

```
1  apply plugin: 'java'
2  dependencies {
3     compile fileTree(dir: 'libs', include: '*.jar')
4  }
5  sourceSets {
6     main {
7        java {
8           srcDir 'src' }
9     }
10    test {
11       java {
12          srcDir 'test' }
13    }
14 }
```

Figure 2.21: Example of a Gradle file

Chapter 3

The Dolphin Language

In this chapter we describe the Dolphin language, and associated developments in support of Dolphin, the IMC DSL and the Neptus Groovy plugin. We present Dolphin by providing an overview (Section 3.1), followed by a more detailed explanation of the language features in detail (Section 3.2). We then discuss the functionality of the IMC DSL (Section 3.3) and the Neptus Groovy plugin (Section 3.4).

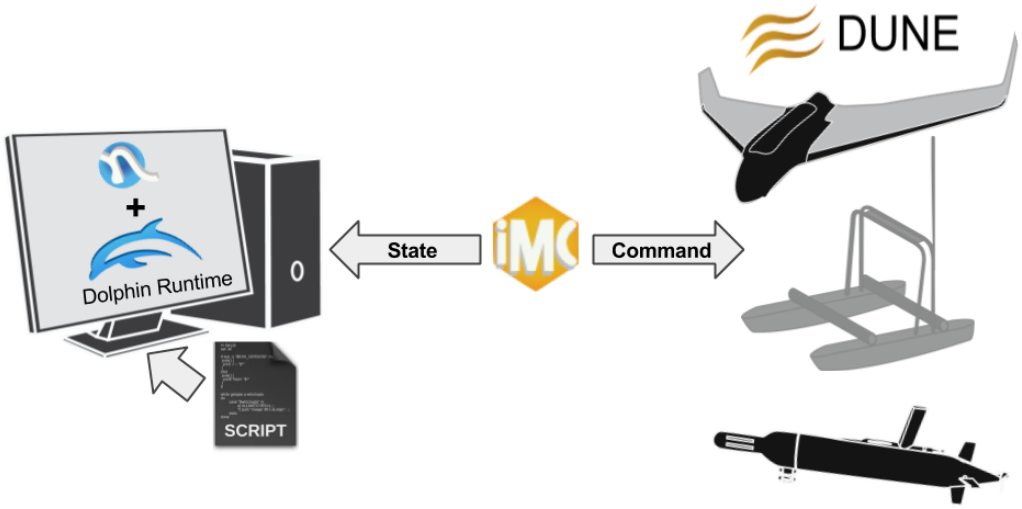


Figure 3.1: Dolphin Architecture Overview.

3.1 Overview

Dolphin is a domain specific language to coordinate autonomous vehicles networks using the LSTS toolchain to interact with these heterogeneous systems. The coordination includes commanding and monitoring the execution of tasks in multiple vehicles, alongside with tasks definition and their allocation to selected vehicles.

3.1.1 Architecture

The integration with LSTS toolchain made possible to abstract the lower level of the network systems, in particular the interaction with autonomous vehicles, specifying their behaviour through a common communication protocol which is understood by all nodes. Dolphin architecture overview is

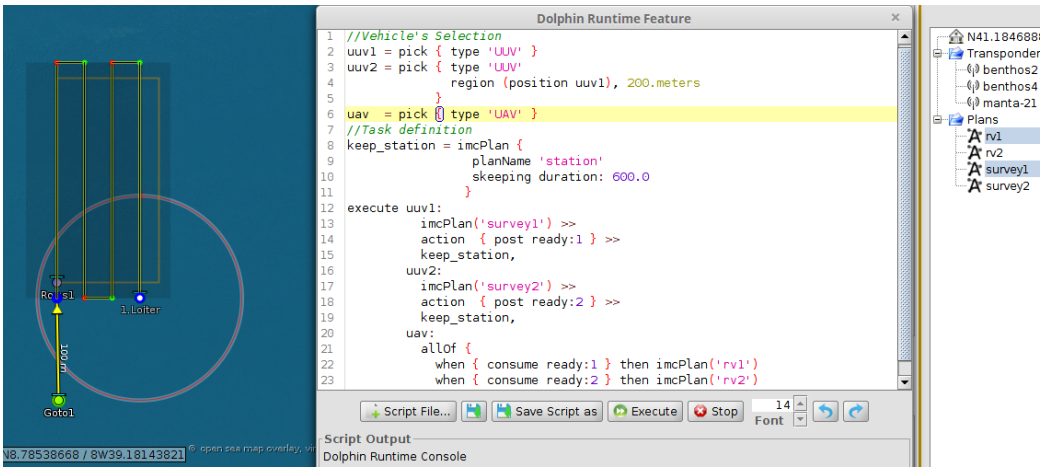


Figure 3.2: Dolphin plug-in in Neptus.

shown in Figure 3.1 where we can identify three main sections:

1. Dolphin programs can be edited and executed within Neptus, as illustrated in Figure 3.2 which shows Neptus console interface that includes the Dolphin script editor. The figure depicts the use of the Dolphin plugin for edition and execution of programs (shown right), integrated with other Neptus functionality such as IMC plan definition (shown left).

2. Dolphin programs trigger the execution of tasks by autonomous vehicles using the IMC protocol, more specifically through IMC plans that define a sequence of vehicle maneuvers.

3. IMC plans are executed onboard vehicles using DUNE as usual.

Dolphin works "out-of-the-box" in integration with the LSTS toolchain. For integration with Neptus, Dolphin support takes form as a self-contained Neptus plugin, and no changes are/were necessary in IMC or DUNE to run Dolphin programs. Moreover, a "stand-alone runtime" version may be used independently of Neptus, e.g. from the command line or in integration with other software.

3.1.2 Example

To provide a glimpse of the Dolphin language, we present an example program in Figure 3.3. The program at stake is a variation of the rendez-vous NVL program (discussed earlier in Section 2.3). The meaning of the program is as follows:

- The program begins with the selection of three sets of vehicles: uuv1, uuv2 and uav (lines 2-6 in the script in Figure 3.3). The selection is made by type defining different requirements for each one of the sets, except for the second AUV which has to be in a 200 meters radius from the previously selected AUV.

- The program follows with the definition of a task (lines 9-12) in the form of an IMC plan, defined using the IMC DSL (Section 3.3). This IMC plan, keep_position specifies a maneuver to keep the vehicles in a stationary position during 10 minutes (600 seconds).

- The remaining part of the script defines how the execution is made, assigning explicitly each selected vehicle to composed tasks. Next, the program expresses the assignment of different tasks to each of the three vehicles selected earlier, using an **execute** instruction. The tasks are defined by composing IMC plans, like keep_position defined earlier or other IMC plans just referred by an identifier (assuming they are already defined in Neptus and/or a vehicle), using task composition operators.

- In terms of the expressed behavior, it is similar for both of the AUVs. Each executes a survey (lines 15 and 19), signals the end of the survey afterwards (lines 16 and 20) and finishes with the execution of the keep_position IMC plan (lines 17 and 21). In turn, The UAV stays idle waiting for the AUVs to finish their survers. It engages in an individual rendez-vous with each of the AUVs, as soon as completion signals are received (in any order) by the AUVs (lines 24 and 25), executing the plan associated to the finished survey. A possible execution timeline for the example is illustrated in Figure 3.4.

```
1  //Vehicle's Selection
2  uuv1 = pick { type 'UUV' }
3  uuv2 = pick { type 'UUV'
4                region (position uuv1), 200.meters
5              }
6  uav  = pick { type 'UAV' }
7
8  //Task definition
9  keep_position = imcPlan {
10                  planName 'keep_position'
11                  skeeping  duration: 600.0
12                 }
13
14 execute uuv1:
15           imcPlan('survey1') >>
16           action  { post ready:1 } >>
17           keep_position,
18         uuv2:
19           imcPlan('survey2') >>
20           action  { post ready:2 } >>
21           keep_position,
22         uav:
23           allOf {
24                   when { consume ready:1 } then imcPlan('rv1')
25                   when { consume ready:2 } then imcPlan('rv2')
26           }
```

Figure 3.3: Example script - Rendez-vous scenario

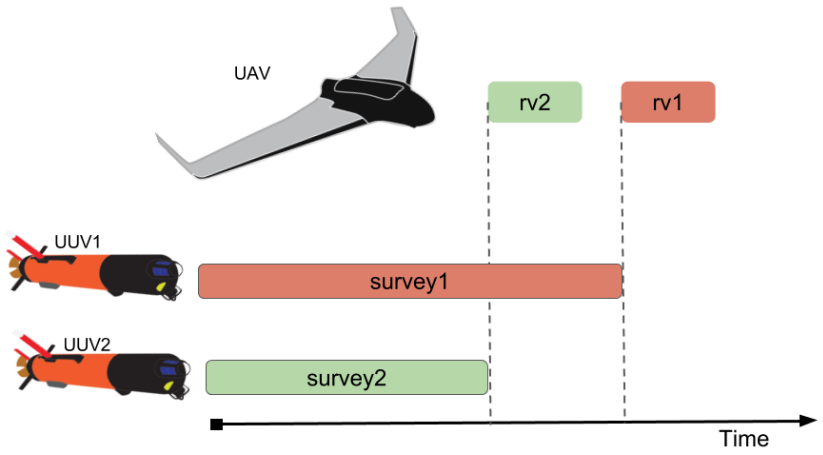


Figure 3.4: Execution diagram example in time

3.2 Language features

After introducing the overall expressiveness of Dolphin through an example, we now explain in more detail the main features the language.

3.2.1 Vehicle Selection

In a Dolphin program, a fundamental aspects is vehicle selection and task execution. We need to select vehicles for subsequent task execution. For instance, consider the following simple example:

```
v1 = pick { type 'UUV' }
v2 = pick { type 'UAV' }
```

The code above express the selection of two vehicles using the **pick** instruction. Besides the vehicle type (UUV or UAV), other criteria can additionally be specified for vehicle selection using **pick**, as listed in Table 3.1. For instance, in the previous example of Figure 3.3, we used the region in combination with type.

Table 3.1: Vehicle selection criteria for **pick**

Method	Description	Default	Example
region	Geographical region	any	region <Location>, 1.km
id	Vehicle's Name	any	id 'lauv-arpao'
count	Number of vehicles	1	count 2
payload	Payload Requirement	any	payload 'Camera'
timeout	timeout for vehicle's selection	∞	timeout 2.minutes
type	Type of the vehicle	any	type 'UUV'

3.2.2 Task allocation and execution

Assuming we have selected vehicles v1 and v2 using **pick**, we can order the execution of tasks to each of them using the **execute** instruction, as follows:

```
execute v1: imcPlan('waterSurvey'),
        v2: imcPlan('airSurvey')
```

In this code, the tasks are IMC plans that are assumed to be stored in Neptus (when using the Neptus plugin) or the vehicle (using the stand-alone runtime). Alternatively, we can use the IMC DSL (Section 3.3) to define the IMC plans programmatically. For instance, we can have:

```
execute v1: imcPlan{
                planName 'waterSurvey'
                ... // maneuvers
                },
        v2: imcPlan('airSurvey')
```

Coming back to the original example, we should note also that the code is a shorthand for:

```
execute  imcPlan('waterSurvey')[v1] |
         imcPlan('airSurvey')[v2]
```

where [ ] is the operator to allocate tasks to vehicles set and | is the operator for concurrent execution. Other operators can be used beyond [] and | to define tasks compositionally, like the >> operator which specifies sequential execution. We discuss other operators later in the text. For instance:

```
execute (imcPlan('waterSurvey1') >> imcPlan('waterSurvey2'))[v1] |
        imcPlan('airSurvey')[v2]
```

which is equivalent to:

```
execute v1: imcPlan('waterSurvey1') >> imcPlan('waterSurvey2'),
        v2: imcPlan('airSurvey')
```

Both fragments determine the execution of waterSurvey1 followed by waterSurvey2 at v1 while v2 concurrently executes airSurvey.

Anonymous vehicle allocation

As discussed previously, the [] operator assigns tasks to vehicles explicitly. The allocation may alternatively be "anonymous", if we do not use the operator at all. For instance, consider:

```
execute  imcPlan('waterSurvey') | imcPlan('airSurvey')
```

In this case programmers do not control the allocation, since it is made anonymously using any previously selected vehicles. The allocation is in this case made by the language runtime, that uses the heuristic of allocating vehicles according to the proximity of their position with the reference position of the first maneuver in each IMC plan that can be executed. The anonymous allocation does not take into account the vehicle type since IMC does not have that kind of information. This limitation could invalidate the example above since it requires different types of vehicles. If the selected vehicles were from the same type they could be easily deployed in this type of scenarios.
Another way to made the IMC plans allocation anonymously is:

```
vehicles = v1 + v2
Task = imcPlan('waterSurvey') | imcPlan('airSurvey')
execute Task[vehicles]
```

Generally, the allocation assigns vehicles sets to tasks that can be simple IMC plans or their composition. This example demonstrates the operators used with the vehicle sets where a + b represents union, a & b intersection, a − b set difference and a | { closure } represents a restriction to the set.

3.2.3 Event-based task operators

As we saw in the previous section, tasks can be used individually or combined with operators, composing them concurrently:

```
t1 = imcPlan('waterSurvey')
t2 = imcPlan('airSurvey')
```

Table 3.2: Event-based operators

Type	Syntax	Restriction
Time Limit	during { <time>} run <Task>	Executes the task during the specified amount of time
Event-Flow	until { <Condition>} run <Task>	Executes the task until the <Condition>is satisfied
Event-Flow	waitFor { <Condition>} then <Task>	Starts the task execution when the <Condition> is satisfied

Choice	choose {
	when { <Condition>}
	then <Task>
	when { <Condition>}
	then <Task>. . .
	}
		Executes **one** of the tasks in a
		then block if the corresponding
		condition in
		the when block is satisfied

Choice	allOf {
	when { <Condition>}
	then <Task>
	when { <Condition>}
	then <Task>. . .
	}
		Executes each task in a
		then block in turn as soon as
		the corresponding condition in the when block
		is satisfied

```
execute t1 | t2
```

using | operator, or sequentially using **the** >> operator:

```
t1 = imcPlan('waterSurvey1')
t2 = imcPlan('waterSurvey2')
t3 = imcPlan('airSurvey')

execute (t1 >> t2) | t3
```

This is not the only possible way to define tasks compositionally. Event flow operators, listed in Table 3.2, can also be defined for more expressive programs. These operators coordinate task execution according to triggers that can be time intervals, events or logical predicates.

For instance, event-based operators can be used to express dependence to a condition, holding execution of another task while a condition is not satisfied:

```
execute waitFor {
            p1 = position v1
            p2 = position v2
            p1.distanceTo p2 > 200.meters
                 }
         then t1
```

or doing the opposite, that is, running a task up until a certain condition is satisfied (possibly interrupting the task execution):

```
execute until {
            p1 = position v1
            p2 = position v2
            p1.distanceTo p2 <= 200.meters
                 }
         run t1
```

The choice operator is made up of **when condition then task** blocks, causing only one of the tasks to execute in the then block as soon as the condition in the corresponding **then** block is satisfied. Assuming that p1 and p2 are vehicles positions, it can used as follows:

```
choose {
        when { p1.distanceTo p2 > 200.meters } then t1
        when { p1.distanceTo p2 > 500.meters } then t2
        }
```

The **allOf** operator in turn, specifies a full set of tasks to be executed when the correspondent condition is satisfied:

```
allOf {
        when { p1.distanceTo p2 > 200.meters } then t1
        when { p1.distanceTo p2 > 500.meters } then t2
      }
```

In support of event-based behavior, some simple (non-composed) tasks can be defined:

- an **action** A to be performed, in particular A may be a **post** tag:value action that adds tag:value to a global event queue;
- a **condition** C to wait for, in particular C may be a condition of the form **consume** tag:value that consumes an event from the global event queue, if one is defined with the specified tag and value;
- **idle** t where t is an amount of time delays execution for the specified

amount of time t;

For instance, these may be used as follows:

```
execute condition {
        p1 = position v1
        p2 = position v2
        p1.distanceTo p2 > 200.meters
        } >> action { post trigger:1 }
```

3.3 The IMC DSL

The IMC DSL can be used to generate IMC plans programatically. The IMC DSL can be embedded in Dolphin scripts (as discussed earlier) but also in scripts programmed using the Neptus Groovy plugin (Section 3.4), or possibly other contexts (it merely depends on the base IMC library).

3.3.1 Example

We demonstrate the usage of the IMC DSL to define a task in a Dolphin script in Figure 3.5. In the presented script the built plan specification consists in a sequence of maneuvers Goto and Loiter (see maneuvers description in 3.3.2). Global parameters such as the plan name, speed, depth (z) and location are defined at the beginning, followed by the maneuvers themselves. Each maneuver uses the current location, speed and z in the plan if they are not re-defined in the arguments. Payload usage can also be defined in the arguments as a list as shown in the line 16 of the script. The move method is an alternative way of define the current location of the plan, and is used to define the location of the next invoked maneuver (lines 15 and 17 of the script).

3.3.2 Features

Maneuvers

The following IMC maneuvers [24] are supported by the IMC DSL:

- Goto: Go to a specified location (waypoint), finishing the maneuver upon arrival.

```
// Pick an UUV

APDL = location 41.18456, -8.70590
v = pick {
  type 'UUV'
  region APDL, 2.km
  id 'lauv-noptilus-3'
}
// IMC plan defined programatically
t2 = imcPlan {
    planName 'waterSurvey'
    speed 1.5, Speed.Units.METERS_PS
    z       0.0, Z.Units.DEPTH
    locate Location.APDL
    move 30,-125
    goTo payload:[[name: Camera]]
    move (-30,-50)
    loiter()
}
execute v:t2
```

Figure 3.5: Building an IMC plan specification in a Dolphin script.

- Loiter: Go to a location and stay there moving around the waypoint in a specified vertical reference and with a defined radius during the specified amount of time.
- Station Keeping - Go to a location and stay at surface for a defined amount of time.
- Launch: Go to a certain location after detecting that the vehicle is in the water.
- YoYo: Similar to Goto but in this case traversing the water column between two defined depths (yo-yo or saw-tooth pattern).
- Popup: Go to the surface at a certain location until the vehicles position is corrected through GPS in the defined time interval.
- Compass Calibration: The calibration of the compass is normally made to compensate the variation in the magnetic field at the region which

the vehicles operate.

- Rows: Survey longitudinally a rectangular area with a defined horizontal step.

Payload Activation

We are able to define payload requirements for each maneuver of the plan, activating their usage in the associated actions of the maneuvers (during calibration phase). Some parameters associated to these payloads could also be defined in the DSL.
All these features mentioned above result in a single IMC plan specification according to the current version of the protocol used between the systems.

3.4 The Neptus Groovy Plug-in

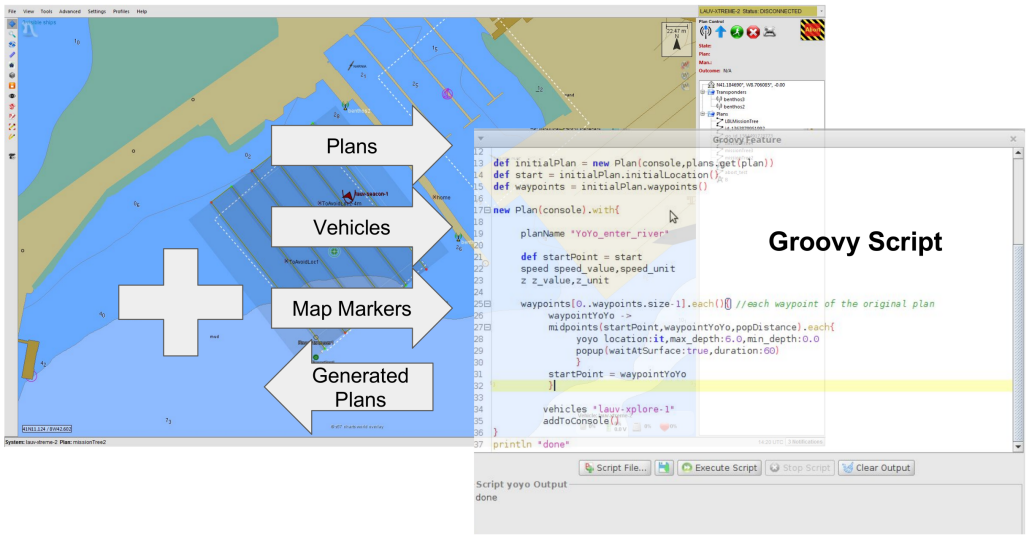


Figure 3.6: Groovy Plug-in Overview.

The incorporation of Groovy in Neptus began with a plug-in which was created to script plans, having access to some defined console information. The loaded information origins from the current mission being used in Neptus console and is mapped in already defined variables in the scripts and

represents all the plans, available vehicles and the points of interest of the map.
In the plug-in scripts we can manipulate already defined plans or generate ones using the IMC DSL introduced before in Section 3.3. Normally, this manipulation is made through the waypoints which can be deployed to calculate distances and to define intermediate waypoints.
In Figure 3.6 we depict the plugin's overview, illustrating a text editor for the scripts in which we can use variables that bind to Neptus data (for plans, vehicles and map markers). The scripts can modify this data, in particular it may be used to generate new plans.

3.4.1 Example

In Figure 3.7 we present a script written in Groovy that be used in the plug-in to list the bindings variables data. We print the data related to each variable by using Groovy's control flow method: `eachWithIndex` (lines 3, 9 and 15). This method iterates over collections (maps in this case) in a functional programming fashion.
All the output generated by the scripts (lines 1, 5, 11 and 17) are redirected to the plug-in's console, an output panel located in the bottom of the window (see Figure 3.6).

3.4.2 Features

Binding Variables

We can access, in the Groovy scripts, binding variables representing some information in the Neptus console such as:

- Plans associated with the current mission being used in the console.
- Points of Interest that are marks in the current console map.
- Vehicles in service represented in a list and updated during the execution of the script.

Neptus usage

Besides the explicit information loaded in the script from the current Neptus console with the bindings variables, existing classes in Neptus source are also

```
println "Binding Variables"
if(vehicles!=null){
        vehicles.eachWithIndex {
                vehicle,index ->
                        println index+".VEHICLE: "+vehicle.key
                }
        }
if(plans!=null){
        plans.eachWithIndex {
                plan,index ->
                        println index+".PLAN: "+plan.key
                }
        }
if(locations!=null) {
        locations.eachWithIndex {
                loc,index ->
                        println index+".LOCATION: "+loc.key
                }
        }
```

Figure 3.7: Groovy script to list Neptus console bindings.

used in the Groovy engine. This usage allows taking advantage of the available tools and features and more classes can be easily added by customizing the groovy engine used to run the scripts.

Groovy Standard Library

All the expressiveness from the Groovy programming language can be deployed in these scripts which gives more flexibility and possibilities codifying scripts. In the script in Figure 3.7 we depict such usage in the deployed methods as println or eachWithIndex mentioned before.

Chapter 4

Design and Implementation

In this chapter we describe the Dolphin language implementation. We begin with the architecture details in Section 4.1, followed by the listing of tools used in the development of the language in Section 4.2. We present a description of each of the identified components of the core library in Section 4.3.We proceed with the description of the IMC DSL in Section 4.4, the details of the language implementation in Neptus (Section 4.5) and in the stand-alone version (Section 4.6). We finish this chapter presenting the details of the Groovy plug-in implementation in Section 4.7.

4.1 Architecture

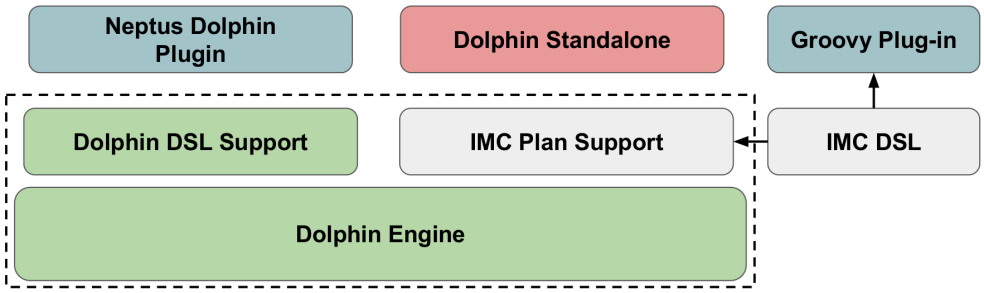


Figure 4.1: Dolphin implementation architecture

In Figure 4.1 we depict the architecture of the language implementation

where we can identify the following components:

- The language common runtime that served as the bases for the implementations made in Neptus and on a stand-alone version.
- The integration with Neptus was made extending the software functionalities by means of a plug-in.
- The stand-alone version was implementing on an IMC protocol runtime, serving as the language platform.
- The language has native support for IMC plans, represented as tasks, being capable to generate plans specifications within a script using the IMC DSL developed.

The main difference between the two versions of the implementation is the creation and management of the IMC protocol instance. Being in the first case, handled by a pre-existing Neptus package having to interact with that API. On the stand-alone version, the creation and usage is all managed in the Dolphin implementation, increasing the workload.
In the figure is also depicted the complementary work that resulted in the Groovy plug-in in Neptus. The plug-in is used for scripting existing plans or generated ones by the IMC DSL embedded, using other users input and binding information from the Neptus console.

4.2 Development tools

The following tools were used to support the language design and implementation:

- **Groovy and Java programming languages** - these two Java platform languages were used in the implementation of Dolphin since one of the objectives of this thesis was a better integration with LSTS toolchain, namely Neptus and IMC which are written in Java. This choice enabled direct integration with the toolchain code. In particular, Groovy features (described in Section 2.5) were also used in the DSL design and are described later on this chapter (see Section 4.3.2).

- **Maven** - the code for the core of the language is structured as a Maven[1] project, allowing for an automated compilation and deployment process.
- **IMC Java library** - the Java implementation[2] of the IMC protocol is used in the IMC common support of the language and in the standalone version to manage communication.
- **RSyntaxTextArea library** - is a library[3] for a text editor with syntax highlighting and code folding for Java Swing applications. It is used in the editor for the Dolphin plug-in in Neptus.

4.3 Core Library

In this section we describe the components of the language core library. This is the library used in the different language implementations, so it consists in the foundations for the language usage.

4.3.1 Common Runtime

Engine

The language engine is responsible for running the scripts having only one instance trough the program execution (corresponding to the singleton pattern). The engine creation requires the definition of an associated platform in which the language is implemented. The scripts are evaluated through a shell provided by the Groovy library. The language syntax is loaded into this shell customizing some compiler parameters as imports or bindings, and evaluating possible extensions files.
The engine is also responsible for pausing and suspending the script execution, handling the exceptions and activating the cleaning mechanisms after the script execution or in case of error. The scripts are executed in a separated thread having a new environment associated to that execution. The environment is in charge of the task allocation, bounding the selected vehicles before the start of the execution.

[1] Maven build system tool website: `http://maven.apache.org`
[2] The IMC Java library is available in `https://github.com/LSTS/imcjava`.
[3] RSyntaxTextArea library website: `http://bobbylight.github.io/RSyntaxTextArea/`

Platform

The language platform is required for the engine creation since it is responsible to bind the language runtime to the environment where it is going to be implemented.
In the platform code fragment presented in Figure 4.2, we can identify the following binds, which the platform is responsible for:

- **Connecting Nodes** - does the conversion of the network nodes into language native nodes, abstracting the way this information is obtained.
- **Task** - in charge of defining the representation of their particular task into the language.
- **I/O operations** - handles the engine output messages and users input data.
- **Language extension** - customizes the engine compilers according to the new functionalities added through the extensions files. It is also responsible for defining the location where these files are placed into the host system, loading them for evaluation.

```
1  public interface Platform extends Debuggable {
2
3    NodeSet getConnectedNodes();
4
5    PlatformTask getPlatformTask(String id);
6
7    void displayMessage(String format, Object... args);
8
9    void customizeGroovyCompilation(CompilerConfiguration cc);
10
11   List<File> getExtensionFiles();
12
13   String askForInput(String prompt);
14 }
```

Figure 4.2: Code fragment of platform interface in Java

As mentioned before, the platform was implemented via two different runtimes: in a Neptus plug-in version and in the stand-alone version using

IMC as the platform. The details of both versions are described later in this chapter. The platform interface implementation of these versions is presented in the UML class diagram on Figure 4.3 generated with ObjectAid [2] extension in the Eclipse IDE.

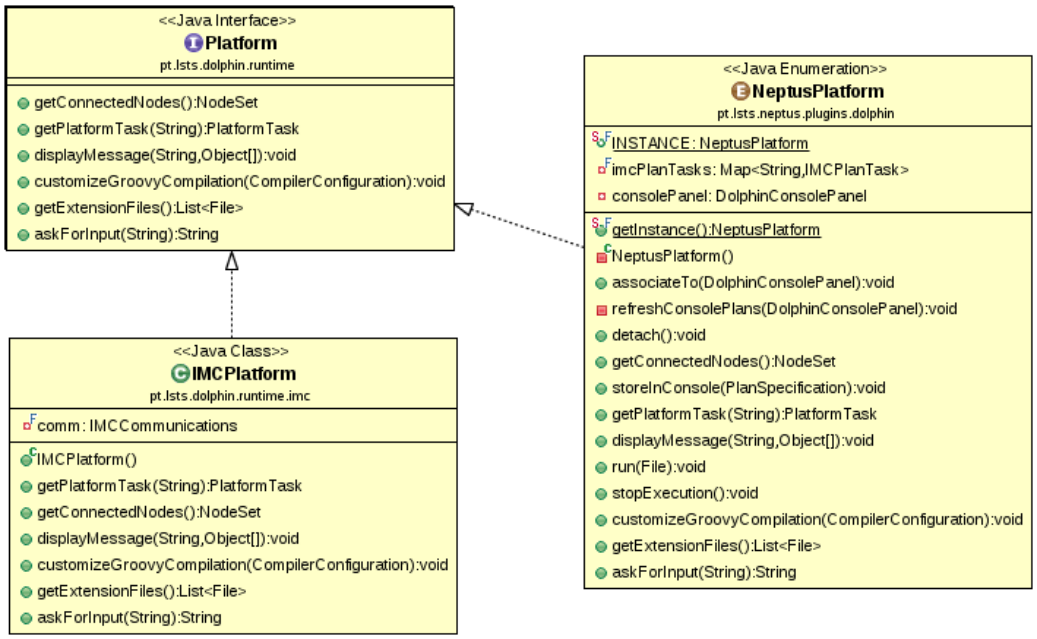


Figure 4.3: Platform class diagram

Nodes

The vehicles are currently the only available nodes in the language. Each vehicle is mapped in a language node having a specific type, payload capabilities, position and an assigned task defined during the allocation. Each node has an associated connection timeout which is by default 20 seconds and can be changed by user defining values between 5 seconds to one hour. After this time is passed, if no information has been reported by the vehicle, the task associated to it is considered to be failed, entering an error state. The vehicles are grouped in sets according to the filters defined in their selection and by using the operators described in Section 3.2.2. These sets can be explicitly or anonymously used in the task execution.

Tasks

The language basic type of tasks are presented in Figure 4.4 (IdleTask, ConditionTask, PlatformTask and ActionTask). These types mirror the elementary tasks presented previously in Section 3.2.3 and they can be combined using operators creating more complex tasks. In terms of composition, these tasks can be arranged sequentially or concurrently originating respectively SequentialTaskComposition and ConcurrentTaskComposition type of tasks. The execution of these composed tasks is made by decomposing the task into elementary types of tasks, defining the allocation and execution order for each one of the originating tasks. The IMC plans are one specific implementation of the platform type of task, since it depends on the communication protocol understood by the systems being used.

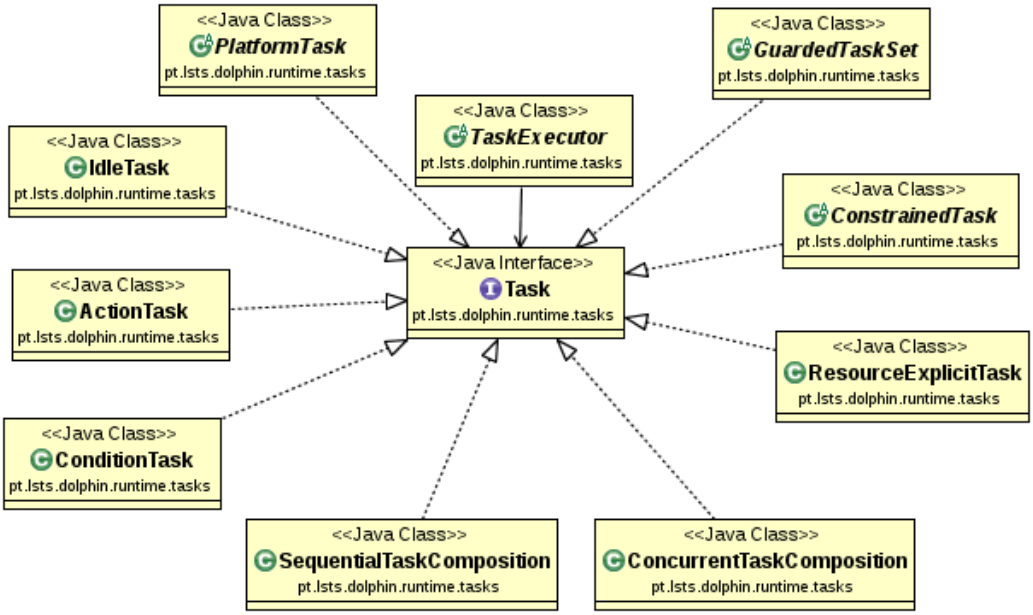


Figure 4.4: Task class hierarchy diagram

Choice operators origins a set of guarded tasks each one bounded by the associated condition. On the other hand, restriction operators usage origins tasks that are limited by a predicate or by time (ConstrainedTask).
Each task defined in the language has an associated executor to monitor their execution. Besides the executor, the task implementation has to specify how its allocation is done to the available set of nodes. In the particular case of

the resource specific type of tasks, the allocation is done according to the specified node set.

TaskExecutors

Each elementary task have an associated TaskExecutor to control their execution, having different restrictions according to the type of task. The TaskExecutor is responsible for starting and finishing the task execution. In each step of the task execution, it verifies the restrictions and defines the state accordingly.
In the diagram presented in Figure 4.5 the different extensions of TaskExecutors, where we can identify:

- SimpleTaskExecutor - controls the execution of the most elementary task types.
- ConstrainedTaskExecutor - monitors the condition associated to the task in each cycle of the engine. If the condition is satisfied, it considers the task completed.
- PlatformTaskExecutor - takes into account the platform particularities and information represented in the correspondent type of task.

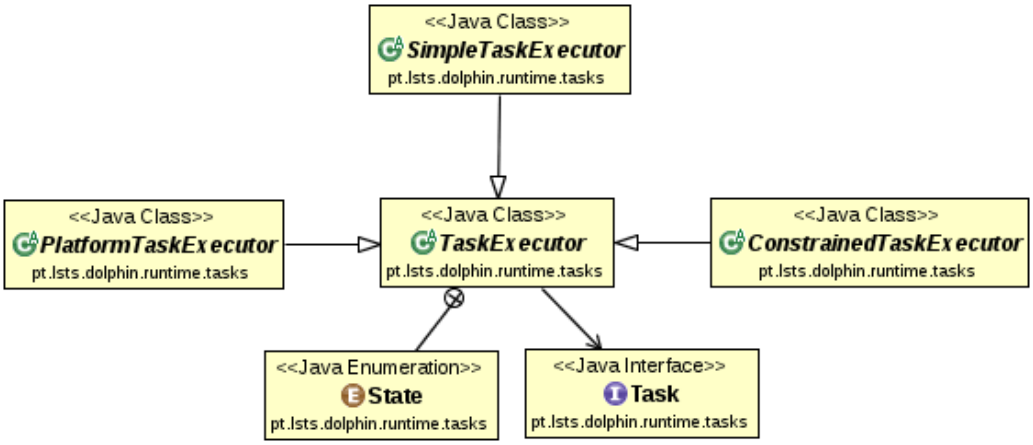


Figure 4.5: TaskExecutor class diagram

4.3.2 DSL Support in Groovy

Dolphin DSL benefits from Groovy languages features as the flexible syntax, operators overloading and MetaClasses. In this section we describe how these features were used in the design of the language syntax. All the languages instructions implementations uses the `@DSLClass` annotation created to statically load them during the compilation. The instructions which receive closures as arguments have an associated builder to construct the language types.

Operators

The operators presented in Section 3.2.2 were implemented through Groovy operators overloading feature [12], defining the behaviour according to the supported types. Groovy enables the usage of a group of symbols in the DSL syntax by implementing their correspondent method.

Signals

Signals are used as trigger to events and they are implemented by a native variable type created in the language runtime utilities. The list of signals manipulation methods is presented in Table 4.1.
The difference between **poll** and **test** is that the first also supports closures as argument which allows the definition of more complex predicates. All the signals are implemented through the **test** which is used by the TaskExecutor in each cycle of the engine to verify if the variable associated with the signal has changed their value.

Table 4.1: Signals manipulation

Syntax	Description
consume <Event>	Consume an signal
post <Event>	Post a signal with an associated value
poll { <Event>}	Verifies the value associated to the signal
test <Event>	Verifies the value associated to the signal

Other instructions

Besides these basic instructions presented in the previous chapter in Section 3.2, there are more definitions that can be made by the user such as the timeout for communications with Nodes after which the connection is considered broken or locations definition using coordinates (corresponding to **setConnectionTimeout** and **location** instructions). Other users input/output can also be defined using the **ask** and **message** instructions, being the last one replicated in a panel in the Dolphin plug-in. There are also instructions to control the language engine such as the **halt** instruction to interrupt the script execution and the **pause** instruction to suspend it during some period of time.

Units

Another Groovy feature used was the MetaClass [5] which allows the addition of properties to basic Java classes and types such as String or Number. This feature was used to add different units to the Number class and their respective conversion metrics as we can see in table 4.2, exemplifying their usage in the line 11 of the script in the Figure 3.3. Percentage units are also supported without the symbol, being converted to the correspondent decimal number.

Table 4.2: Dolphin Supported Units

Units	Supported Conversions	Standard
Angle	Degrees, Radians	Radians
Distance	Meters, Kilometers	Meters
Time	Seconds, Minutes, Hours, Days	Seconds

4.3.3 IMC plan Support

IMC plans are one particular basic type of task of the language since it represents the platform being used in the implementation. Therefore they are treated as a platform type of task, mapping only part of the information contained in the IMC message into the Task type. The remaining information is kept through the plan specification and corresponds to the IMC protocol implementation details and properties. One example of such property is the

payload information present in the maneuvers specifications. This information is preserved and used during the task allocation, filtering vehicles that satisfy the plan requirements.

4.4 IMC DSL

The IMC DSL was embedded in both implementations of the language through extensions, using the instruction **imcPlan**. It was used to concisely define IMC plan specifications within a program, represented as tasks. The DSL was also implemented in Groovy for the same reasons it was used for Dolphin implementation.
The supported maneuvers (see Section 3.3) are implemented through their correspondent type in the IMC Java library. The maneuvers parameters are defined in a map in arguments benefiting from Groovy flexible syntax. If no arguments are defined in the maneuvers, the currently defined plan parameters, namely the location, speed and z are used. Other particular parameters associated to maneuvers have to be explicitly defined in the argument map in the form:
`maneuver <parameter>: 'value'.`
There is also support for specifying the vehicles which the plan was built for, although no verification is provided by IMC to check maneuvers compatibility to the vehicles. The IMC plan specification is then built from the maneuvers defined in the DSL along with their transitions, if it passes the IMC validation method.

4.5 Dolphin Neptus Plug-in

The language was implemented by means of a plug-in, extending Neptus and adding the core library to it. In this case the IMC protocol usage was done by an internal manager instance in Neptus, not having to configure the library implementation as in the stand-alone version.

4.5.1 Plug-in Architecture

Dolphin plug-in works as an extension to the Neptus console, having the following stages in the implementation:

- **Dolphin core library implementation** - the language semantics are implemented adapting to the Neptus environment. The syntax can be extended through additional files that are loaded by the engine through the platform.
- **Plug-in description and layout** - here the console is extended defining some parameters in the `@PluginDescription` annotation in Neptus. The parameters are used in the Neptus console configurations, used to enable the language plug-in usage.
- **Initial routines** - in this stage the plug-in functionality is called. The language engine is initialized once, first time the plug-in is enabled in the console.
- **Event Handling** - different handlers are listening in the editor through their components listeners resulting from the user interactions. Other events related to the Neptus console were also subscribed that were triggered by modifications in the plans database in console.

4.5.2 Neptus Platform

The platform can be associated to the currently open console in Neptus being detached in case the console is closed. It serves as an intermediate between Neptus and Dolphin. This is where the additional compiler configurations are made, defining the extensions files that are evaluated by the engine. It keeps the information related to current console, interacting with the plug-in editor to execute requests from the language runtime. Along with the platform there is the implementation of the languages Node sets, IMC tasks and the associated executor adapted to Neptus. The implementation maps existing types in Neptus into Dolphin types.

4.5.3 Editor

The plug-in editor functions as a overlay console to the Neptus mission console, implementing the RSyntaxTextArea text editor mentioned before. Its design is presented in Figure 4.6, where we can identify an embedded text editor which is associated to the currently open console in Neptus, being uncoupled from the engine in case the console is closed. The text editor is used to load and edit the language scripts that are by default in a directory defined

through the @NeptusProperty annotation in Neptus which is configurable by users. There is also a panel to show the engine generated output information (bottom of the editor in Figure 4.6). Users inputs are introduced in a pop-up window when the **ask** instruction is used on the script.
The execution can be controlled through the start and stop buttons in the console. There are also buttons related to the script edition allowing to save the current changes, define font size and manipulate the modifications stack (undo and redo).

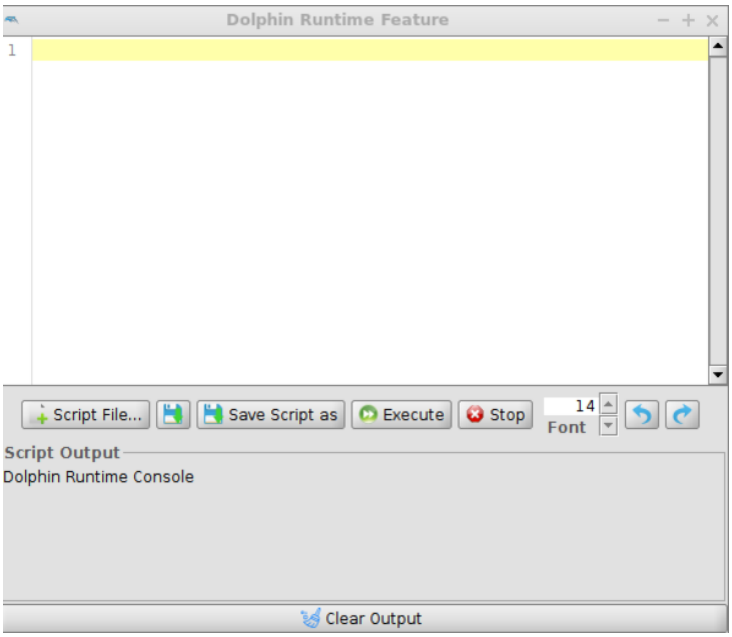


Figure 4.6: Dolphin plug-in editor in Neptus

4.5.4 Extensions

In the Neptus DSL implementation, additional script instructions can be configured from extension files, loaded during the platform creation, which affects the engine customization. The **imcPlan** instruction is added to the language syntax in this stage and it can be either an IMC plan specification generated by the IMC DSL or fetched from the current Neptus console plans database. More functionalities as functions or instructions can be statically added to the language in these extension files.

4.6 Standalone IMC Runtime

The main difference between the stand-alone implementation and the one made in Neptus was the different IMC protocol implementations that got used. In this version there is an extra effort regarding the management of the network communication with the systems, which implied the creation, access and removal of the protocol instance.
In terms of GUI, this version runs on the terminal having some helper scripts to start the execution of already defined programs. During the execution the users do not have a visual perception of the network state, as happens in Neptus, but there is still some feedback information about the execution state generated by the engine and printed in the terminal. Users input are also available through the console, using the instruction **ask**. Language extensions can also be defined in this version, being implemented in the same way as described before in Section 4.5.4.

4.7 Groovy Plug-in

Since Neptus is an easily extended software developed in Java, we began the integration of the Groovy language by creating a plug-in that could run scripts in the Java environment. The fact that Groovy also runs in the Java Virtual Machine (JVM) greatly simplified embedding of this scripting environment into Neptus (only requires the addition of the Groovy language library). In order to run complex Groovy scripts we had to use an engine in the Java code that can be configured with parameters as imports, bindings or compilation strategies [12]. All these parameters are customizable in the engine configurations that is used to run the scripts.
The adopted strategy in the plug-in implementation was to incrementally add and test Groovy language features into Neptus console. The Neptus console information, namely plans, available vehicles and the points of interest used in the scripts were implemented through script variable bindings. It is a mechanism that allows the usage in Groovy of defined variables in Java without declaring them. These variables can be accessed in Groovy scripts as long they are not declared in the script, otherwise they overwrite the previous information.

Chapter 5

Experimental Results

This chapter presents experimental results from field tests conducted in order to evaluate the Dolphin language, primarily, but also the IMC DSL and the Neptus Groovy plugin script. The tests were conducted using real UUVs, in some scenarios complemented by a simulated UAV, due to constraints on the availability of this type of vehicle at the time of tests. They took place in three distinct locations and in the context of other LSTS operations:

1. Preliminary field tests conducted at the entrance of Leixões harbour, APDL, for a first validation of Dolphin (Section 5.1);
2. Field tests in open sea near Tróia, during the REP'17 exercise organised by LSTS and the Portuguese Navy (Section 5.2);
3. Field tests in Douro's river mouth, specifically to validate the Groovy Neptus plugin features and the context of the DRiP project to study Douro's river plume (Section 5.3);

For each of these field tests, we present the overall planning that took place, the subject of testing, the most relevant/complex scenarios that were exercised for the tested functionality, and the corresponding results. The results at stake result from post-processing vehicle logs after execution of tests using the Mission Review and Analysis (MRA) feature of Neptus and other complementary scripts.

5.1 APDL field test

The first field tests with Dolphin took place at APDL (Administração dos Portos do Douro, Leixões e Viana do Castelo) on June 30, 2017. The location at stake is the port entrance located between Leça da Palmeira (top/north) and Matosinhos (bottom/south) beaches in Porto, shown in Figure 5.1a, where LSTS regularly conduct tests.
The vehicles used on this mission were:

- LAUV-Noptilus-1, equipped with a Sidescan Sonar and a Multibeam Sonar;
- LAUV-Noptilus-2, equipped with a Sidescan Sonar;
- LAUV-Noptilus-3, equipped with a Sidescan Sonar, a Multibeam Sonar and a Camera;
- X8-02, a simulated UAV.

All the UUVs used in these tests are also equipped with a Forward-Looking Echo Sounder to detect obstacles. The UUVs are shown in Figure 5.1b. Also shown another UUV, LAUV-Arpao which was used in some concurrent tests by LSTS unrelated to Dolphin.

This was the first time the language was tested in a real operation scenario although it was under a controlled environment. All tests had to have the vehicles moving on sea surface because the runtime was assuming a timeout of 20 seconds. Even though the vehicles are able to send their state using the acoustic modem, this did not include the plan identifier information which was required to know which plan was being executed. As such, the acoustic reports were ignored and only information sent via Wi-Fi was used for these tests.

5.1.1 Mission timeline

The tests had an incremental level of complexity in order to validate the language features and primitives separately and then combined. The objective was to facilitate the debugging in case of error or an unexpected behaviour. We began with some dry tests where the execution was intentionally expected to fail. After the vehicles were deployed on the water we started with simpler tests commanding the execution of both plans fetched from the Neptus

(a) APDL test location

(b) UUVs used for testing at APDL

(c) Overview of surveys plans for rendez-vous scenario

Figure 5.1: APDL field test description

console and plans generated by the IMC DSL (Section 3.3) built in the Dolphin scripts. After these preliminary tests of the language, a small change

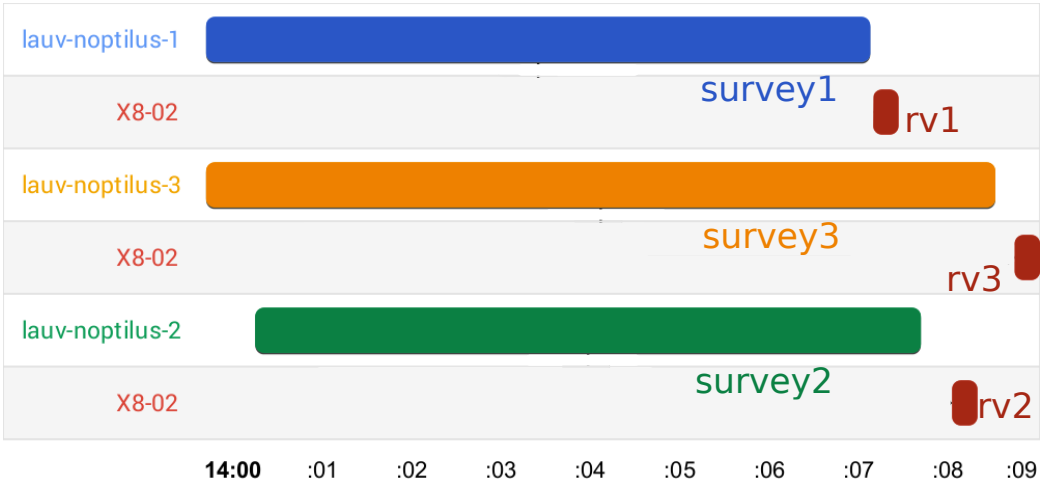
was made, adjusting the language timeout for connection with the vehicles to real operation conditions since they are required to stay submerged for a reasonable amount of time. This parameter became user-defined, having a minimum of 5 seconds and at most one hour wherein the default value was set to 20 seconds. Once the adjustments were made, we conducted the vehicle rendez-vous scenario discussed below.

5.1.2 Rendez-vous scenario reviewed

Once again we used the UUV-AUV rendez-vous scenario, based on the NVL example (Section 2.3) and introduced in the Dolphin overview (Section 3.1) presented earlier. In this, each of the three UUV conducts a survey in a different area (see Figure 5.1c), after which they engage in a rendez-vous with an UAV. The Dolphin version allows this scenario to be more expressive, in sense that the rendez-vous order is not that strict, and a rendez-vous may start without need for all surveys to complete.
The corresponding Dolphin program is shown in Figure 5.2a. It begins with the vehicles selection restricted only by the required type (lines 1-4 in the script), followed by the execution composition. The execution is divided in three concurrent phases, each one of them associated to a selected vehicle. The composition of each survey includes the plan execution (lines 7,10 and 13 in the script in Figure 5.2a) and the signal posting to indicate the end of the execution (lines 8,11 and 14 in the script). The **allOf** instruction verifies if any of the surveys have been executed, consuming the respective signal and triggering the rendez-vous plan execution.

```
uuv1 = pick { type 'UUV' }
uuv2 = pick { type 'UUV' }
uuv3 = pick { type 'UUV' }
uav  = pick { type 'UAV' }

execute uuv1:
          imcPlan('survey1') >>
          action { post ready:1 },
        uuv2:
          imcPlan('survey2') >>
          action { post ready:2 },
        uuv3:
          imcPlan('survey3') >>
          action { post ready:3 },
        uav:
          allOf {
                  when { consume ready:1 } then imcPlan('rv1')
                  when { consume ready:2 } then imcPlan('rv2')
                  when { consume ready:3 } then imcPlan('rv3')
          }
```

(a) Script



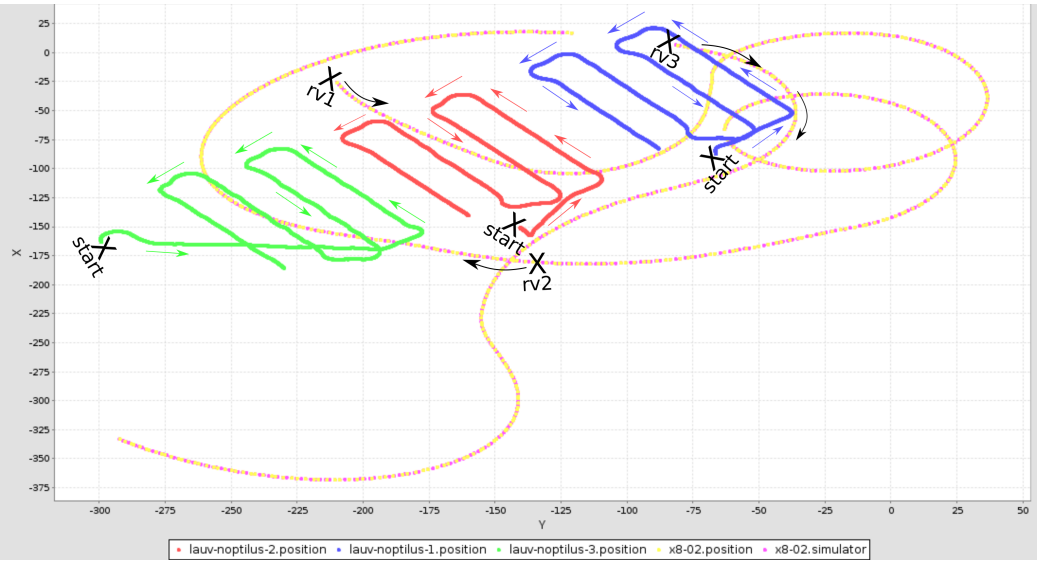
(b) Execution timeline

Figure 5.2: Rendez-vous scenario program

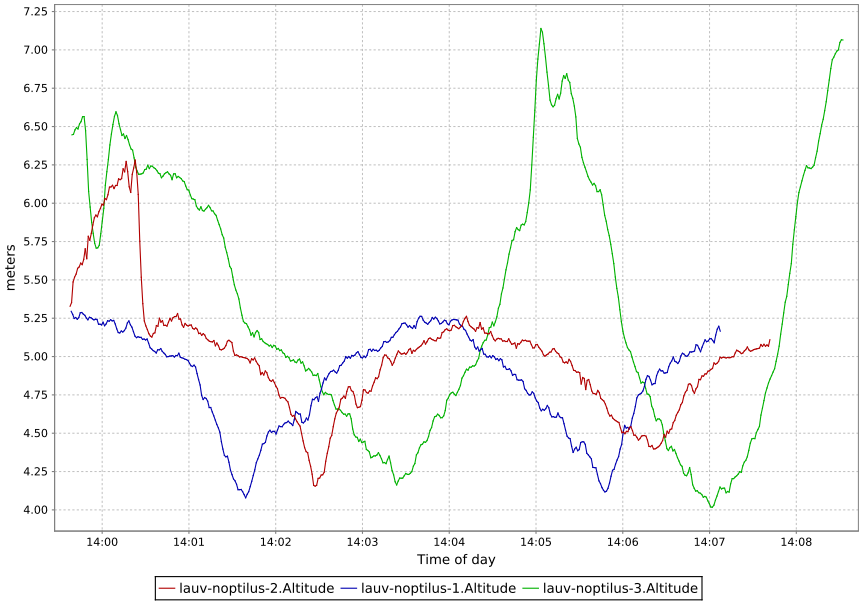
5.1.3 Results of the script execution

In the Figures 5.2b to 5.4 we present the processed data from the script execution obtained from the vehicle's and the language runtime logs:

- In the timeline presented in Figure 5.2b we can notice which vehicle executed which plan, since the vehicles selection was made only by type.
- In Figure 5.3a is visible the UUVs positions during the surveys execution overlapped by the UAV positions (dashed) during the rendez-vous.
- We present the Z axis variation of the UUVs, in the form of the altitude values in meters, during the surveys execution in Figure 5.3b. In this plot, the blue color corresponds to the *lauv-noptilus-1*, the green corresponds to the *lauv-noptilus-3* and the red one corresponds to the *lauv-noptilus-2*.
- In Figure 5.4 we presented the processed data from the multibeam sonar used during the survey in *lauv-noptilus-1*. The data collected corresponds to the bathymetry of the survey area (4 to 6 meters deep), which mirrors the values from the vehicle in the altitude plot presented above since the UUVs only operated at surface.



(a) Rendezvous scenario - Vehicle's XY Plot



(b) Rendezvous scenario - Vehicle Altitude Plot

Figure 5.3: Rendezvous scenario - Vehicle's execution plots

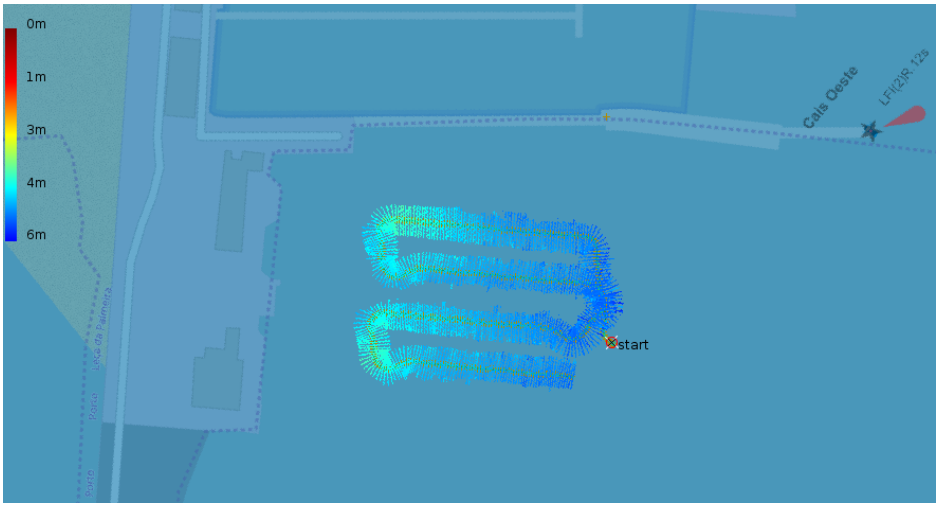


Figure 5.4: Rendezvous scenario - lauv-noptilus-1 multibeam sonar bathymetry data from Neptus

5.2 REP'17 tests

5.2.1 Context

The Rapid Environment Picture is an annual exercise organized by LSTS and the Portuguese Navy, and includes the participation of other invited partners. The main objective of this yearly exercise is to demonstrate and test new operational concepts in real scenarios. This year's edition took place at the Tróia peninsula with some technological, scientific and security/defence goals[1] as:

- Mixed-initative planning and execution, whose objective was to demonstrate deliberative planning on shoreside and onboard defining the behaviour of multiple vehicles.
- SaVeL - Sado Estuarine Outflow centred in the collection of hydrographic data to calibrate and validate numeric models of the Sado Estuarine and coast.
- SNoW - Sado Non-linear Internal Waves that was an attempt to observe internal waves by deploying multiple systems (UAV and UUV).

The Dolphin tests were involved in the multi-vehicles coordination test fields and took place at July 10/11, 2017 at the sea in front of Comporta beach (operation area marked in Figure 5.5a), on board the NRP Cassiopeia vessel (visible in the Figure 5.5b).
The vehicles used in both days of operation were:

- LAUV-Noptilus-1, equipped with a Sidescan Sonar and a Multibeam Sonar;
- LAUV-Noptilus-2, equipped with a Sidescan Sonar;
- LAUV-Xplore-1, equipped with an environmental probe measuring temperature, salinity, pH and redox.
- X8-03, a simulated UAV, as a physical UAV could not be allocated to our tests due to operational limitations at the time

In these operations the Dolphin language was tested in a richer and more challenging scenario than in APDL (Section 5.1). Additionally, we intended to validate the new features added to the language since the last tests. The added features were the possibility of using the vehicles' positions as event triggers, and allowing anonymous allocation to proceed by distance according to the first waypoint of an IMC plan, i.e., selecting the vehicle that is closer to that waypoint if there is more than one available vehicle to execute the task represented by the IMC plan.

5.2.2 Mission timeline

The test were conducted during two days having the same incremental strategy, in terms of complexity, as mentioned before:

- **Day 1**
  The strategy adopted for these tests was very similar to the ones made previously, starting with simple scripts that intentionally failed and moving towards more complex primitive combinations that required events and synchronization. Although this time we are able to test plans with the vehicles moving underwater due to the new feature added at the time. This time we had scripts with timeout for communication set from 20 seconds (default value) up to 10 minutes.

[1]More details and results from this year's REP can be found at `http://rep17.lsts.pt`

- **Day 2**
  As result of the first tests, the Dolphin language anonymous allocation of the node set of vehicles was now made by distance, selecting the node that was closer to the first waypoint of the plan to be executed. The objectives for the second day of operations was to test the new features, anonymous allocation and usage of the vehicle's position, and try a more expressive version of the rendezvous scenario script adding flow control with the **until** instruction.

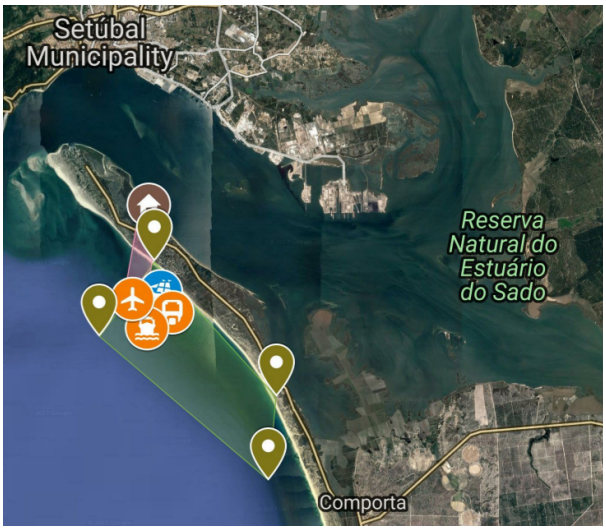
In Figure 5.5c are shown the surveys overview related to the most relevant tests made during these two days of operation and described in the next sections, being the west side of the Tróia peninsula located on the right of the surveys. The surveys were composed by two Rows maneuvers separated by one Popup[2] which allowed to fix the vehicles position in case they were dragged by the tide. The Popup also allowed the Dolphin language runtime to update the state of execution since we discarded the acoustic reports.
The results of both days of operations were positive since we had the expected outcome. We present bellow the most relevant results corresponding the more complex script tested in each day of operation. The other tested scripts are shown in the Appendix A.

5.2.3 Execution synchronization using events

This particular script involved three UUVs in which one of them acted as a master while the other two vehicles executed surveys in response to its inputs (slaves). The idea was having the master execute sequentially two tasks, and, as each of the tasks finished, trigger the execution of tasks in the slave vehicles. Thus, the slave vehicles only started task execution in response to an event generated in association to the execution of the master vehicles.

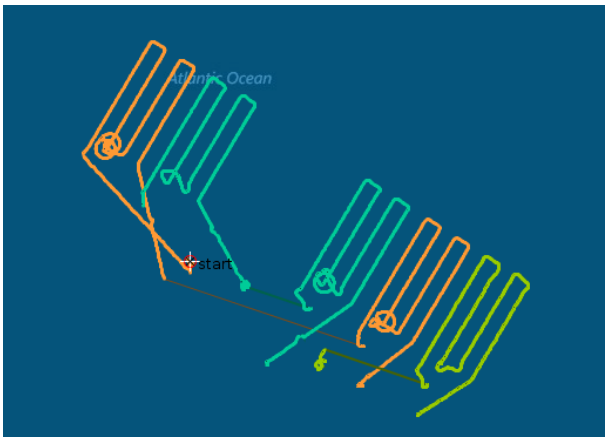
The corresponding program, shown in Figure 5.6a, exposes the language support for event and synchronization. From the vehicles selection in the script (lines 1-9 in Figure 5.6a) above we can notice that the vehicle lauv−xplore−1 executed plan1 and plan2, behaving as master and lauv−noptilus−1 and lauv−noptilus−2 executed respectively survey6 and survey7 behaving as slaves.

[2]The maneuvers sequence are shown in the individual vehicles timelines in Appendix C.



(a) Exercises location

(b) NRP Cassiopeia and some autonomous vehicles used during REP'17

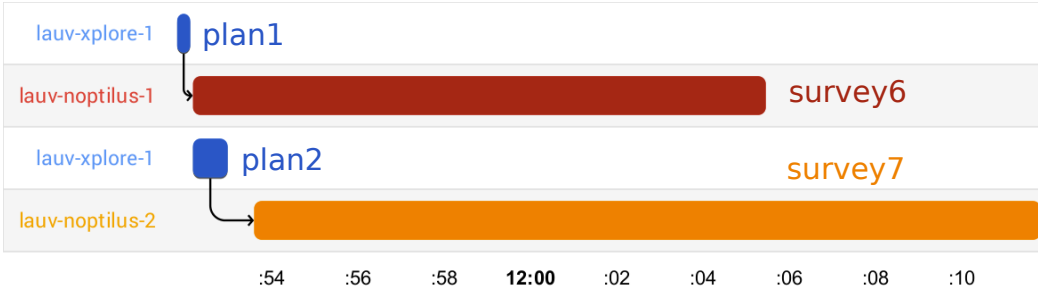


(c) Overview of surveys plan

Figure 5.5: REP'17 exercises description

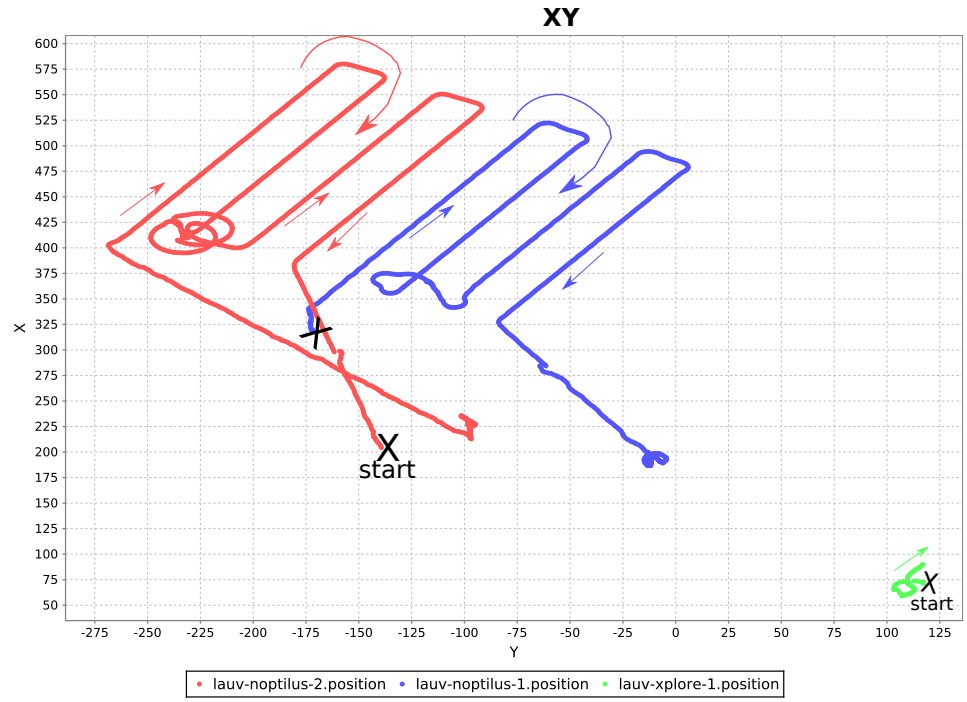
```
1  master = pick { type 'UUV'
2                  id 'lauv-xplore-1'
3                }
4  slave1 = pick { type 'UUV'
5                  id 'lauv-noptilus-1'
6                }
7  slave2 = pick { type 'UUV'
8                  id 'lauv-noptilus-2'
9                }
10
11 setConnectionTimeout  7.minutes
12
13 execute master :
14               imcPlan('plan1') >>
15               action { post x:1 } >>
16               imcPlan('plan2') >>
17               action { post x:2 },
18         slave1:
19               condition { consume x: 1} >>
20               imcPlan('survey6'),
21         slave2:
22               condition { consume x: 2} >>
23               imcPlan('survey7')
```

(a) Script

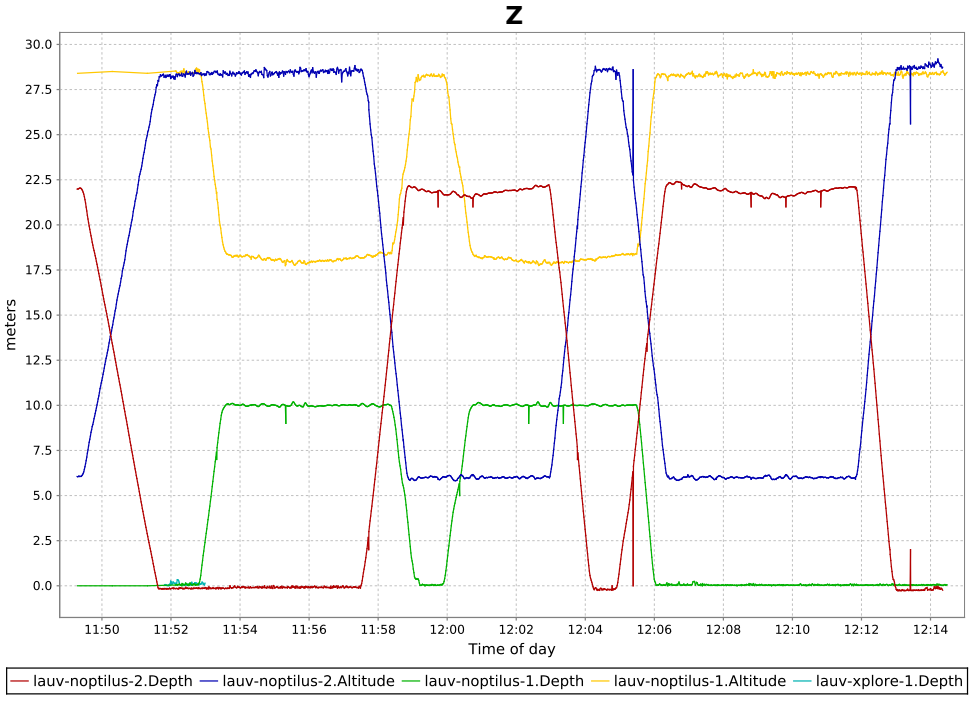


(b) Execution timeline

Figure 5.6: Events and synchronization



(a) Vehicle's positions plots



(b) Vehicle's Z plots

Figure 5.7: Events and Synchronization - plots

5.2.4 Rendez-vous scenario with more combined features

After the first approach made before (see Section 5.1.2), we revisited the rendezvous operation scenario[14] adding more expressiveness than the previous tests made at APDL(Section 5.1), which included maintaining the UAV loitering in a certain position until one of the UUV finished a survey.

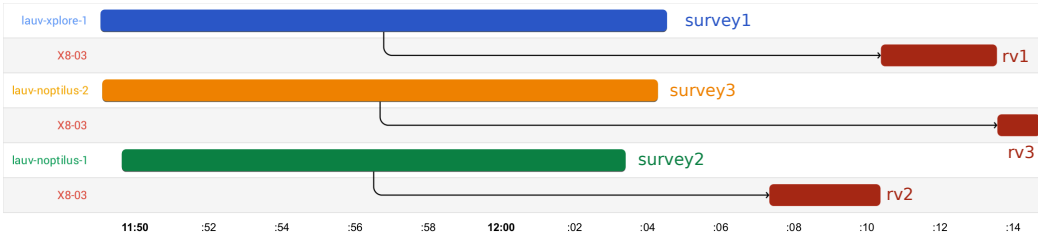
In REP'17, we executed a variation of the rendez-vous scenario recreated in the APDL field tests (Section 5.1.2), experimenting language features that were introduced in the meanwhile. The differences between the script presented before at the Figure 5.2a and the script in Figure 5.8a are:

- The UUVs selection was made at once, specifying the required type and the number of vehicles (lines 1-3 in the script in Figure 5.8a).
- There was a defined connection timeout of 10 minutes (line 7 in the script in Figure 5.8a), since the vehicles executed surveys that required them to submerge (see depth/altitude plot in Figure 5.9b).
- The anonymous allocation was made by distance to the first waypoint of the plan (see Appendix B in Figure B.1), assigning the set of three vehicles to the three concurrent tasks in the execution composition (lines 11-14 in the script).
- The UAV was maintaining loitering until one of the surveys was completed, polling the if one of the correspondent signals were posted. This behaviour has added to avoid the UAV (in simulation mode) to drift, moving away from the operational area.

By the timeline presented in Figure 5.8b we can notice that the survey3 was the first being finished by the UUVs since the rv3 was the first rendez-vous performed. Additionally, the assigned vehicle to execute that survey (lauv-xplore-1 by the timeline) operated only at surface as we can verify in the depth values presented in the plot in Figure 5.9b (red line at the bottom). There was no need to correct the UUV position during the Popup unlike in the other two cases visible in the positions plot in Figure 5.9a, corresponding respectively to the lauv-noptilus-1 and lauv-noptilus-2.

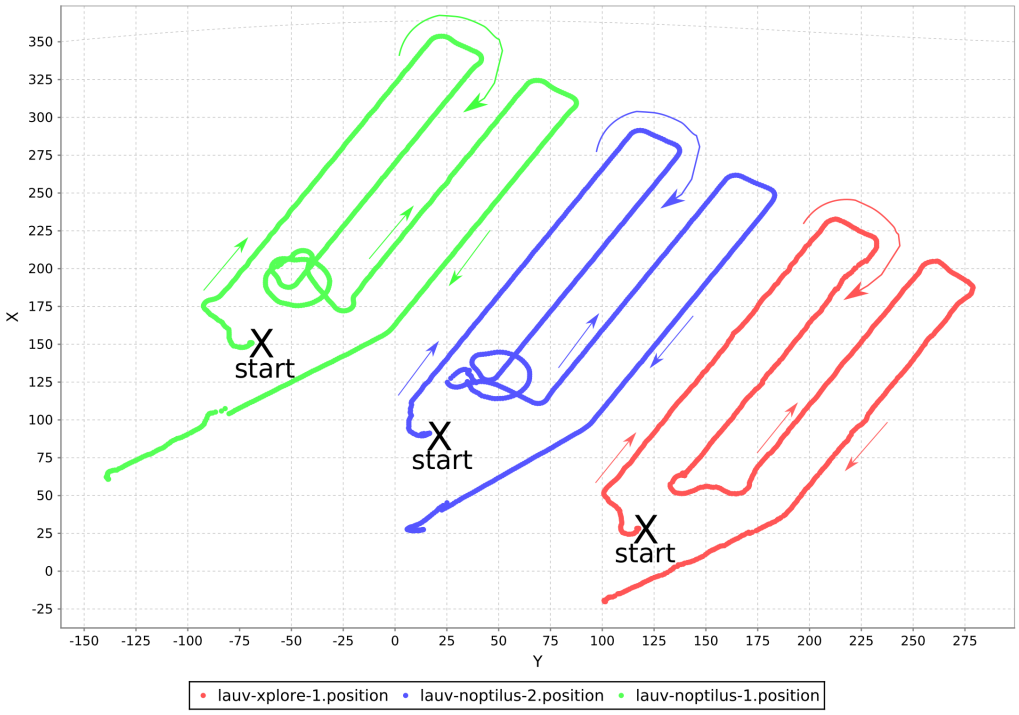
```
uuvs = pick { type 'UUV'
              count 3
            }

uav = pick { type 'UAV' }

setConnectionTimeout  10.minutes

loiter =  until { poll 'ready' } run  stay_still

execute uuvs:
          (imcPlan('survey1') >> action { post ready:1 }) |
          (imcPlan('survey2') >> action { post ready:2 }) |
          (imcPlan('survey3') >> action { post ready:3 }),
        uav:
          loiter >>
          allOf {
                  when { consume ready:1 } then imcPlan('rv1')
                  when { consume ready:2 } then imcPlan('rv2')
                  when { consume ready:3 } then imcPlan('rv3')
          }
```

(a) Script

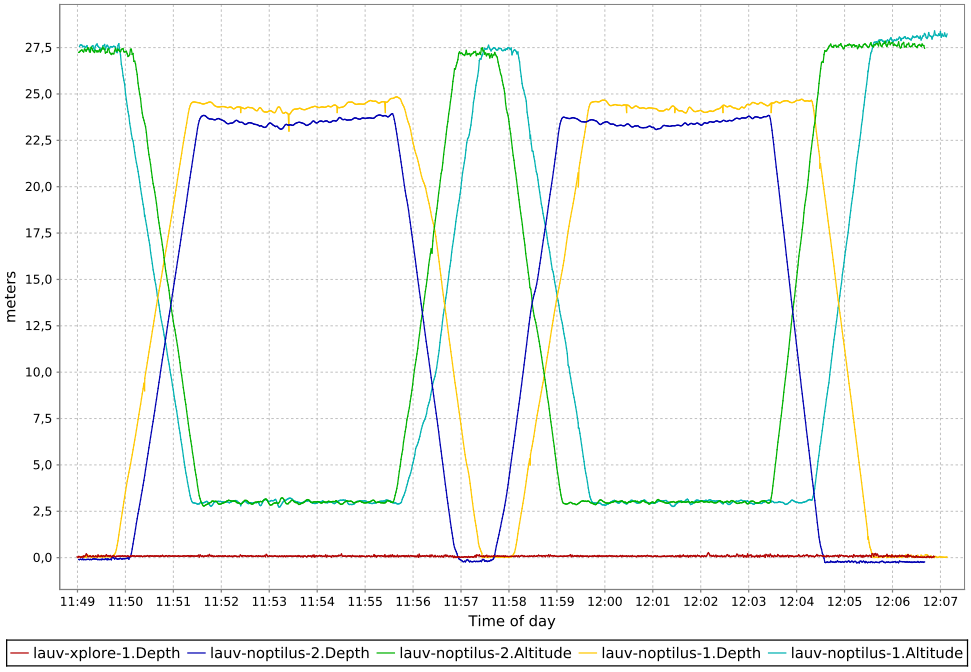


(b) Execution timeline

Figure 5.8: Rendezvous reviewed scenario



(a) UUVs positions plot



(b) UUVs Z plot

Figure 5.9: Rendezvous reviewed scenario - plots

5.3 The Neptus Groovy plug-in

Another result accomplished in the work elaborated for this thesis was the development of a Neptus plug-in that allows using Groovy to interact with the console with bindings for plans, locations and vehicles. More details of the plug-in features and details can be found in the Section 3.4. The re-validation of this plug-in was made on August 13th after some fixes to the previous version and tests. The objective was to validate the automatic generation of plans seizing the opportunity to include them in one of the LSTS's real operation scenarios, the DRiP missions.

5.3.1 DRiP - Douro River Plume tracking

LSTS has been doing missions between the Douro river mouth and the sea trying to identify and collect data associated to the Douro plume. In these missions the vehicles have an onboard executive that tracks the plume while running on an auxiliary CPU interacting with DUNE (Section 2.1.2) on the main CPU via IMC messages (Section 2.1.2). The DRiP executive, starts by moving out of the river's mouth towards the sea until it detects that is outside the fresh water plume (according to a user-defined salinity threshold). After it detects the front of the river plume, it stops, comes at the surface, communicates its position and starts moving again towards the river mouth. In Figure 5.10a, we can notice on the right side the sea and on the left side the entrance to the river channel.
The objective of these missions is to cross the front of the plume several times and, every time it goes out it increases its bearing so that it tracks the position of the front all around the river. Moreover, after a certain timeout is reached, the executive starts executing a predefined plan in order to meet the operators and get recovered. The vehicles used in this mission had a configurable payload defined to collect samples in the water:

- LAUV-Xplore-1, equipped with an environmental probe measuring temperature, salinity, pH and Redox (the vehicle in front in Figure 5.10b);
- LAUV-Xplore-2, equipped with an environmental probe that measures temperature, salinity, chlorophyll and turbidity (the vehicle behind in Figure 5.10b);

(a) Mission Location

(b) Systems used in the August 13 mission

Figure 5.10: DRiP Mission on August 13, 2017

5.3.2 Mission timeline

The mission started with the DRiP parameters definition, followed by the vehicles deployment on the water. The vehicles collected data for 5 hours during which the operators were moving between the river end and the sea according to the water conditions. Since the operators did not have a fixed location by the end of the vehicles operations there was a need to bring them closer to the boat to collect them. Various attempts were made with the Groovy plug-in to generate a plan that made the vehicle approach the boat

using both marks and existing plans on the console. It is important to refer that, since the generated plans were being added to the console, they could be revised by the operators prior to commanding them to the vehicles. This was a great help as it allowed us to iterate through a series of scripts until the expected behaviour was achieved.
At the end some of the generated plans were sent to one of the vehicles through Iridium communication plug-in in Neptus because of their distance to the operators station onboard of a boat.
During one of the vehicles collection, we also tested different maneuvers and their parameters with the other available vehicle, in order to validate other parts of the IMC DSL usage in the plug-in. The generated plan was also sent and commanded the execution via Iridium. When this vehicle was close to the boat we aborted the execution starting the teleoperation to proceed to the recovery.

5.3.3 Plan scripts

To validate the plug-in we tested the generation of a plan that resulted from the manipulation of the waypoints of an existing plan in the Neptus console and other that tested using variable bindings for the generation. The other features of this plug-in were already tested in an early stage of the developments. Namely the vehicles binding variable and other maneuvers from the IMC DSL.

Yoyo-Popup script

Looking at the script presented in the Figure 5.11, we can notice that the manipulated plan fetched from Neptus console was "plan1" which had waypoints in a square path around the area of the boat. The script manipulated these waypoints by translating them to locations to the yoyo maneuver, adding popup maneuvers every 350 meters. In the Figure 5.13 we can verify the vehicles positions during the execution of the plan before aborting it, which is consistent with the generated plan preview available in the Figure 5.12.
The vehicle's depth can be found in the Figure 5.14 along with the salinity data collected during the execution of the generated script.

```
//----Parameters Definition-----

def popDistance    = 350 //in meters
def plan           = "plan1"   // already defined in the console
double speed_value = 1500.0
def speed_unit     = "RPM"
double z_value     = 5.0
def z_unit         = "DEPTH"
if(plans.get(plan)==null)
        println "Could not find initial plan "+plan
def initialPlan = new Plan(console,plans.get(plan))
def start = initialPlan.initialLocation()
def waypoints = initialPlan.waypoints()

new Plan(console).with{

        planName "YoYo_drip"
        def startPoint = start
        speed speed_value,speed_unit
        z z_value,z_unit

    //each waypoint of the original plan
        waypoints[0..waypoints.size-1].each(){
                waypointYoYo ->
                midpoints(startPoint,waypointYoYo,popDistance).each{
                        yoyo location:it,max_depth:8.0,min_depth:0.0
                        popup(waitAtSurface:true,duration:60)
                        }
                startPoint = waypointYoYo
                }

            vehicles "lauv-xplore-2"
            addToConsole()
}
```

Figure 5.11: Script to generate popups between yoyos maneuvers for each 350 meters

Vehicle recovery - first attempt

The first attempt was made when the boat was drifting in the river. We manipulated an already defined plan on the console, using its waypoints as reference to the locations to the yoyo maneuvers, adding intermediate popups every 500 meters and at the originally final waypoint. In the Figure 5.15 we

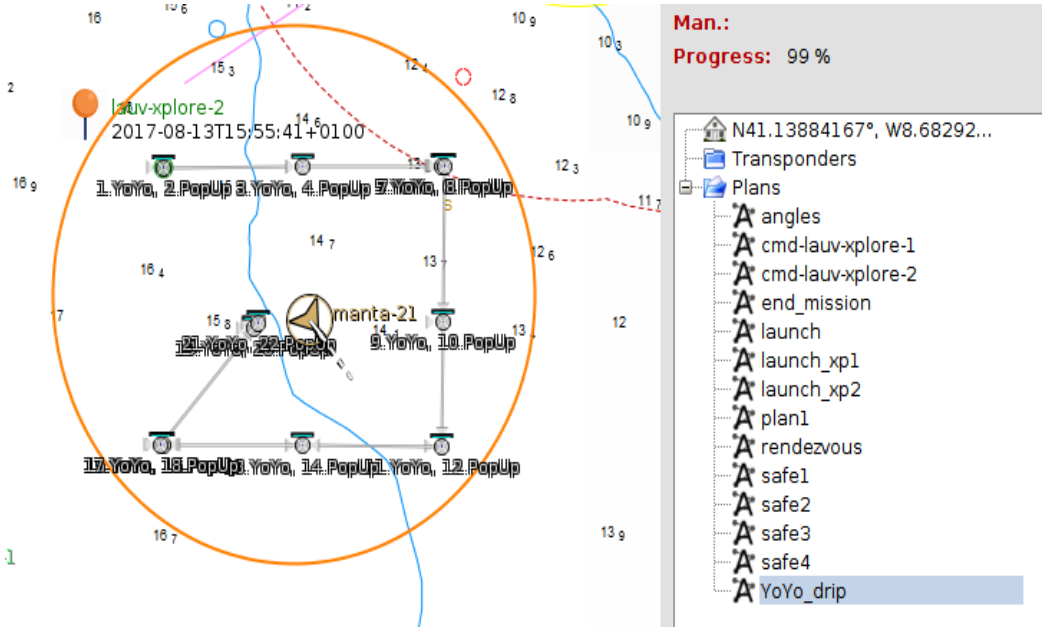


Figure 5.12: Generated plan with yoyos and popup preview in Neptus

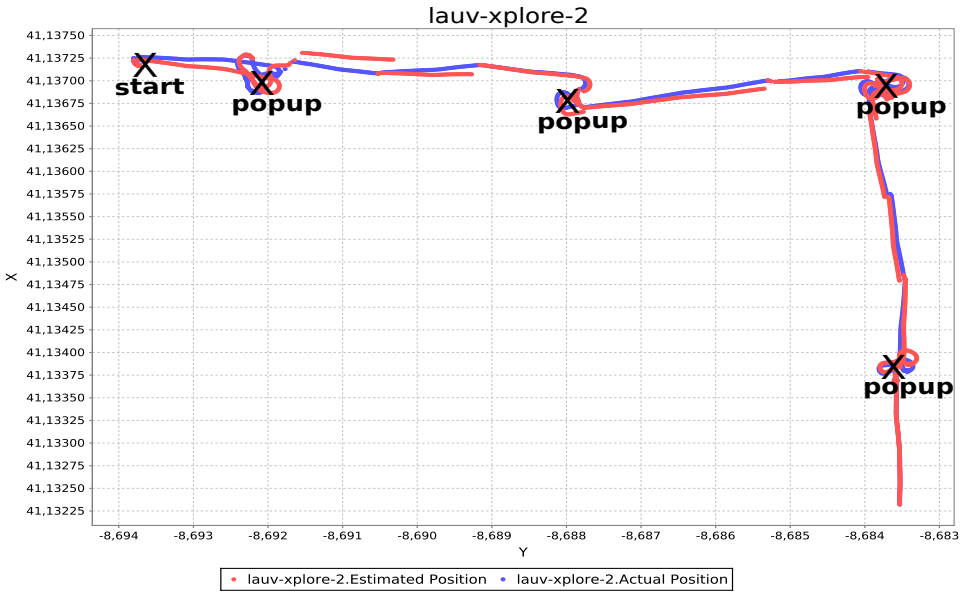


Figure 5.13: Vehicle's position plot

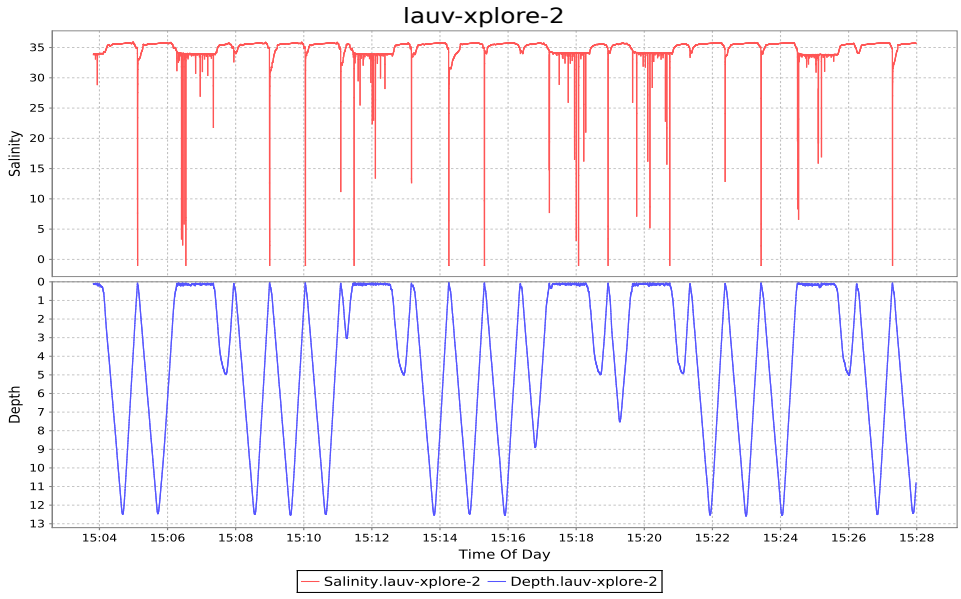


Figure 5.14: Vehicle's logged salinity and depth plot. Notice that salinity as some erroneous data when the vehicle is at the surface due to the sensor being momentarily out of the water.

can notice part of the generated plan preview and the associated script used. The script used to generate this plan can be found in the Appendix D in Figure D.2.

Vehicle recovery - second attempt

In the other script used in these tests the idea was to use the 2 defined marks on the console map to make the vehicle approach the boat area moving in yoyo until the final defined point and doing a popup every 400 meters. One of the marks was in the last known position of the vehicle, reported via Iridium communication and the other one was closer to the area where the boat was drifting at the time the script was made. In the Figure 5.16 we can notice in the inferior left corner the mark from the last known position of the vehicle which was used on the script visible in the plug-in text editor on the right. We can also see the first waypoint at 400 meters from the defined mark and other two waypoints corresponding to the 800 meters and final waypoint respectively. The script for the second attempt is shown in the Appendix D

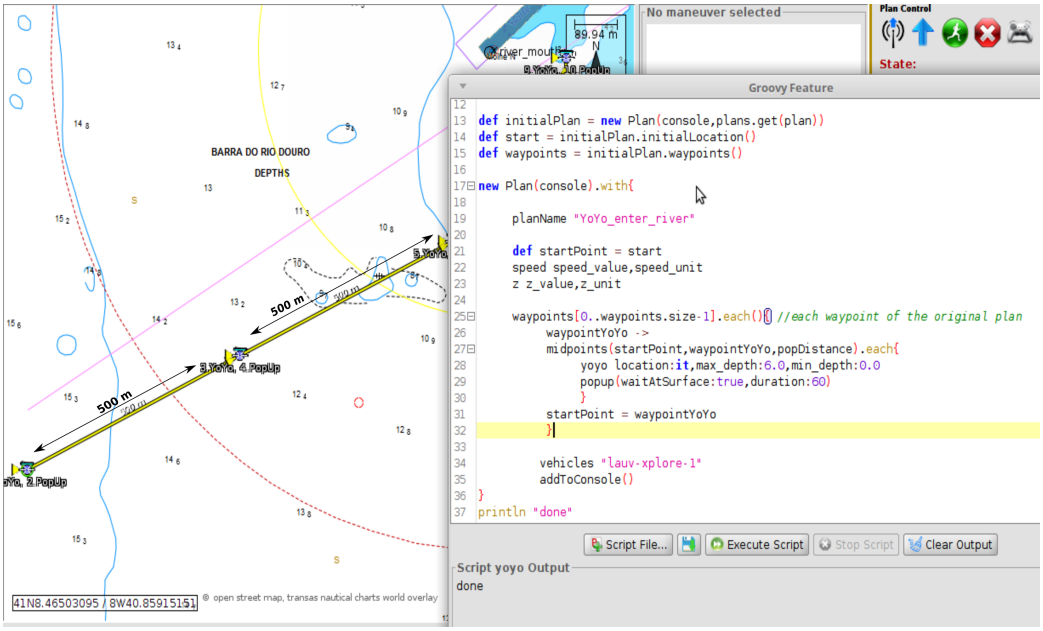


Figure 5.15: Groovy Plug-in: Console Screenshot with the first attempt script and plan preview.

in Figure D.3.

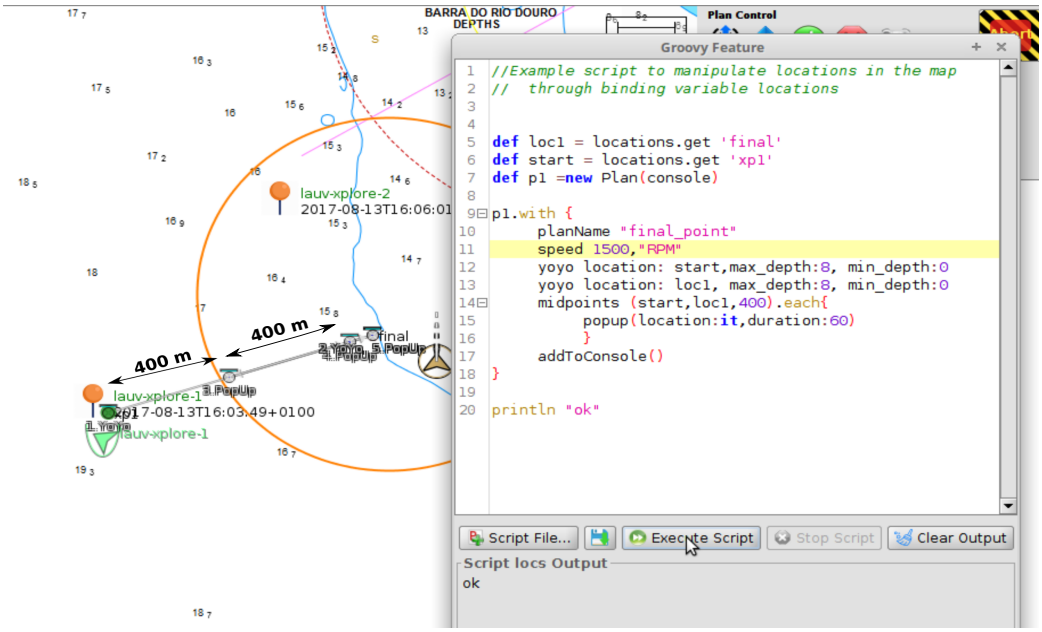


Figure 5.16: Groovy Plug-in: Console Screenshot with the second attempt script and plan preview.

Chapter 6

Conclusions

In this chapter we first summarize the work developed for this thesis (Section 6.1), analysing the limitations and the results obtained from the chosen approach. We then conclude with suggestions of future developments that can be to enhance and continue the implementation (Section 6.2).

6.1 Discussion

We have created the Dolphin language, a DSL that allows programmable coordination of autonomous vehicles networks, integrated with LSTS toolchain. Prior to this thesis, the behaviour of each vehicle had to be manually defined by operators, a dull and error-prone task. With Dolphin, however, the users can create a system-level definition of behaviour where multiple vehicles are tasked automatically according to their availability and capabilities, which greatly simplifies the operation of such networks.
NVL, a predecessor of Dolphin, was a first effort towards this problem. NVL had limited integration with LSTS toolchain and was much less expressive and easy to use than Dolphin.
The language functionalities includes vehicles selection, task specification and their allocation to the selected vehicles for execution. The execution can be composed combining different operators supported by the languages types (Nodes and Tasks). The tasks supported by the vehicles are in the form of IMC plan specification that can be generated using the developed IMC DSL or fetched from Neptus or from the vehicles plans database.
Taking into account that one of our main objectives: integration of the lan-

guage with LSTS command and control software Neptus, we think the implementation benefited from the chosen tools, since both parts belong to the Java platform having the Java Virtual Machine (JVM) in common. Thus, the implementation is extensible and flexible, as demonstrated also by the existence of two different platform implementations, the Neptus-based version and the stand-alone one.

6.1.1 Limitations

During the development and tests of the language we have identified the following limitations, that can be subject to improvements:

- The allocation of a task to a vehicle cannot be changed, once the task starts executing. This leads to no fault tolerance, that could be achieved by migrating the task to another vehicle.
- Error handling in case of task execution failure could be refined, e.g., defining a different behaviour according to the source of the failure.
- Vehicle state inference by the DSL runtime still has limitations when the vehicles use alternative means of communication like Iridium or underwater acoustic communications. This happens because the IMC protocol does not provide normalized information for the different means of communication.
- Verification of vehicles compatibility to IMC plans specification is also a handicap that can lead to failures on the vehicles.
- The language runtime runs in a centralized fashion, a possible source of conflicting (e.g., when multiple CCUs are in use) or erroneous behaviour (e.g. due to network failures).

6.1.2 Evaluation analysis

The following field tests were conducted during the development of this thesis:

- Initially some tests were made in a controlled environment at APDL where it was possible to identify some bugs and test their correction.

- In REP'17 exercises we validated the language in a not controlled environment. Here the communication was very limited, even so Dolphin maintained robust during the communication failures allowing the control of different types of vehicles (bottom-mapping AUVs, oceanographic AUVs and simulated UAVs).

- Validation of the Groovy plug-in in Neptus during a DRiP mission.

We obtain the expected results in most of the field tests since the behaviour was similar to the ones obtained in simulation despite the communication differences between the two environments. These tests served to adjust the implementation according to problems and improvements identified during the mission course or afterwards, analysing the data.
These results shows that is possible to create expressive programs in Dolphin to manipulate multiple autonomous vehicles applied to real operational scenarios.

6.2 Future Work

The Dolphin DSL forms the basics of many possible developments in the future:

- Regarding the level of abstraction of the DSL, human operators could be represented as CCU or ACCU node types since these systems announces their presence in the network and interacts with other systems using IMC. In terms of the vehicles nodes, more filters can be added to the selection as fuel level.

- Multi-vehicle tasks that enable vehicle teams with varying composition also interest us, augmenting the current expressiveness of Dolphin operators for task composition.

- Efforts are being made in the IMC protocol to abstract the communication mean used in the state reported by the vehicles. If this new feature is introduced we believe a better error treatment can be done since only the fully populated reports are taken into account.

- A GUI similar to the one implemented in the Neptus plug-in could be used in the stand-alone version to make it more user-friendly.

- In terms of allocation, it could be done dynamically during program execution instead of the current static allocation made before the execution. This dynamic approach can take into account new nodes inserted in the network or even the replacement of allocated ones in case of failure.

- Letting Dolphin run in distributed manner may be interesting, raising numerous and interesting challenges. For instance, the language could need extensions or shared state and/or node communication.

- Finally, more platforms can be considered as targets for integration with Dolphin beyond the LSTS toolchain, e.g. MAVLink [1] which is a widely used protocol by several UAVs in the market and supported by many robotic platforms like Ardupilot.

Bibliography

[1] MVLink micro air vehicle communication protocol. `http://mavlink.org`. Accessed: 2017-09-23.

[2] ObjectAid uml explorer for eclipse. `http://www.objectaid.com/`. Accessed: 2017-09-25.

[3] J. Borges de Sousa, K. H. Johansson, J. Silva, and A. Speranzon. A verified hierarchical control architecture for co-ordinated multi-vehicle operations. *International Journal of Adaptive Control and Signal Processing*, 21(2-3):159–188, 2007.

[4] L. Chrpa, J. Pinto, M. A. Ribeiro, F. Py, J. Sousa, and K. 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.

[5] F. Dearle. *Groovy for Domain-Specific Languages*. Packt Publishing Ltd, 2015.

[6] C. N. Duarte, G. R. Martel, C. Buzzell, D. Crimmins, R. Komerska, S. Mupparapu, S. Chappell, D. R. Blidberg, and R. Nitzel. A common control language to support multiple cooperating AUVs. In *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*, pages 1–9, 2005.

[7] M. Dunbabin, P. Corke, I. Vasilescu, and D. Rus. Data muling over underwater wireless sensor networks using an autonomous underwater vehicle. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2091–2098. IEEE, 2006.

[8] E. Eberbach, C. Duarte, C. Buzzell, and G. Martel. A portable language for control of multiple autonomous vehicles and distributed problem solving. In *Proc. of the 2nd Intern. Conf. on Computational Intelligence, Robotics and Autonomous Systems CIRAS*, volume 3, pages 15–18, 2003.

[9] A. S. Ferreira, J. Pinto, P. Dias, and J. B. de Sousa. The lsts software toolchain for persistent maritime operations applied through vehicular ad-hoc networks. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 609–616, June 2017.

[10] M. Fox and D. Long. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003.

[11] D. Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[12] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in action*, volume 1. Manning, 2007.

[13] K. A. Kousen and G. Laforge. *Making Java Groovy*. Manning, 2014.

[14] E. R. Marques, M. Ribeiro, J. Pinto, J. B. Sousa, and F. Martins. NVL: a coordination language for unmanned vehicle networks. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 331–334. ACM, 2015.

[15] E. R. B. Marques, M. Ribeiro, J. Pinto, J. Sousa, and F. Martins. Towards programmable coordination of unmanned vehicle networks. In *Proc. IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles*, NGCUV'15. IFAC, 2015.

[16] R. Martins, P. S. Dias, E. R. Marques, J. Pinto, J. B. Sousa, and F. L. Pereira. IMC: A communication protocol for networked vehicles and sensors. In *Oceans 2009-Europe*, pages 1–6. IEEE, 2009.

[17] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL-the planning domain definition language. 1998.

[18] F. Nex and F. Remondino. Uav for 3d mapping applications: a review. *Applied Geomatics*, 6(1):1–15, Mar 2014.

[19] G. E. Packard, A. Kukulya, T. Austin, M. Dennett, R. Littlefield, G. Packard, M. Purcell, R. Stokey, and G. Skomal. Continuous autonomous tracking and imaging of white sharks and basking sharks using a REMUS-100 AUV. In *2013 OCEANS-San Diego*, pages 1–5. IEEE, 2013.

[20] E. Pereira, C. M. Kirsch, R. Sengupta, and J. B. de Sousa. BigActors - A Model for Structure-aware Computation. In *ACM/IEEE International Conference on Cyber-Physical Systems*, 2013.

[21] E. Pereira, C. Krainer, P. M. da Silva, C. M. Kirsch, and R. Sengupta. A runtime system for logical-space programming. In *Proceedings of the Second International Workshop on the Swarm at the Edge of the Cloud*, pages 28–33. ACM, 2015.

[22] E. T. Pereira. *Mobile Reactive Systems over Bigraphical Machines-A Programming Model and its Implementation*. University of California, Berkeley, 2015.

[23] C. Pinciroli and G. Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 3794–3800. IEEE, 2016.

[24] C. d. Q. Pinto. Mixed-initiative planning for networked vehicle systems. Master's thesis, 2017.

[25] J. Pinto, P. S. Dias, R. Martins, J. Fortuna, E. Marques, and J. Sousa. The LSTS toolchain for networked vehicle systems. In *OCEANS-Bergen, 2013 MTS/IEEE*, pages 1–9. IEEE, 2013.

[26] J. Pinto, M. Faria, J. Fortuna, R. Martins, J. Sousa, N. Queiroz, F. Py, and K. Rajan. Chasing Fish: Tracking and control in a autonomous multi-vehicle real-world experiment. In *2013 OCEANS-San Diego*, pages 1–6. IEEE, 2013.

[27] F. Py, J. Pinto, M. A. Silva, T. A. Johansen, J. Sousa, and K. Rajan. Europtus: A mixed-initiative controller for multi-vehicle oceanographic field experiments. In *International Symposium on Experimental Robotics*, pages 323–340. Springer, 2016.

[28] M. A. Ribeiro. Nvl: uma linguagem de coordenação para redes de véiculos autónomos. Master's thesis, 2014.

Appendix A

Dolphin scripts

A.1 Task Operators

```
1 v1 = pick { id 'lauv-noptilus-1' }
2 v2 = pick { id 'lauv-noptilus-2' }
3
4 setConnectionTimeout  120.0 //seconds
5
6 execute  imcPlan('survey4')[v1] | imcPlan('survey5')[v2]
```

Figure A.1: Dolphin script for parallel execution of tasks

```
1 v1 = pick { id 'lauv-noptilus-1' }
2 v2 = pick { id 'lauv-noptilus-2' }
3
4  setConnectionTimeout  30.0
5
6 execute  imcPlan('survey4')[v1] >> imcPlan('survey5')[v2]
```

Figure A.2: Dolphin script for sequential execution of tasks

```
1  ComportaOParea = location 38.43461, -8.86117
2  v = pick {
3    type 'UUV'
4    //region ComportaOParea, 5.km
5    id 'lauv-noptilus-1'
6  }
7
8  // Execute IMC Plan designed in Neptus
9  t1 = imcPlan 'plan1'
10 execute v:t1
```

Figure A.3: Dolphin script usage of IMC Plan from Neptus console as task

A.2 Vehicles Operators

```
1  n = ask 'How many UUVS?'
2
3  v1 = pick {
4          type 'UUV'
5          count n
6  }
7  v2 = pick {
8          type 'UAV'
9  }
10
11 uv = v1 + v2
12 dv = v1+v2-v1
13 iv = uv & v1
14 idF = ask 'Vehicle to filter?'
15 fv = ( (v1+v2) | { v -> !v.getId().equals(idF) })
16
17 message "uv = uv"message "iv = iv"
18
19 message "dv = dv"message "fv = fv"
```

Figure A.4: Dolphin script with vehicles set manipulation

```
n = ask 'How many vehicles?'
uuvs = pick {
        type 'UUV'
        timeout 20.seconds
        count n
        payload 'Sidescan'
        }

setConnectionTimeout  180.0

task_1 = imcPlan 'plan1'
task_2 = imcPlan 'plan2'

message 'Vehicles sucessfully selected: '+uuvs

execute  uuvs: (task_1 | task_2)
```

Figure A.5: Dolphin script with task allocation to set of vehicles

```
v = pick { id 'lauv-xplore-1' }
initialPos = position(v)

task1 = until {
         position(v).distanceTo(initialPos) > 20
       } run imcPlan('plan1')

task2 = during { 30.seconds } run imcPlan('plan2')
execute idle (10.seconds) >> task1 >> task2
```

Figure A.6: Script with vehicles position usage as event trigger

A.3 Failure Tests

- Wrong Selection by Payload Requirement Specification

```
1 uuv = pick {
2
3 type 'UUV'
4 id    'lauv-noptilus-2'
5 payload 'Multibeam'
6 }
7
8 message "Selected UUV uuv"
```

Figure A.7: Dolphin script with wrong selection by payload and vehicle's id

- Wrong Selection by Type of Vehicle

```
1 uav = pick {
2
3 type 'UAV'
4 id    'lauv-noptilus-1'
5 }
6
7 message 'Selected UAV uav'
```

Figure A.8: Dolphin script with wrong selection by vehicle's type and vehicle's id

- Wrong Selection by Area of the Vehicles

```
1 ComportaOpArea = location 38.425711111111106, -8.852072222222223
2 v = pick {
3   type 'UUV'
4   region ComportaOpArea, 2.meters
5 }
```

Figure A.9: Dolphin script with selection by vehicle's region

Appendix B

Dolphin runtime logs

B.1 Runtime anonymous allocation by distance

```
allocate        Vehicles: { lauv-xplore-1 lauv-noptilus-2 lauv-noptilus-1 }
allocate        Considering: lauv-xplore-1
allocate        Considering also: lauv-noptilus-2 (121.204022 < 83.048241 ?)
allocate        Considering also: lauv-noptilus-1 (244.740381 < 83.048241 ?)
allocate        Selected: lauv-xplore-1
allocate        Requirements: [id=,type=,payload=[Sidescan],area=]
allocate        Vehicles: { lauv-noptilus-2 lauv-noptilus-1 }
allocate        Considering: lauv-noptilus-2
allocate        Considering also: lauv-noptilus-1 (153.554003 < 71.832014 ?)
allocate        Selected: lauv-noptilus-2
allocate        Requirements: [id=,type=,payload=[Sidescan, Multibeam],area=]
allocate        Vehicles: { lauv-noptilus-1 }
allocate        Considering: lauv-noptilus-1
allocate        Selected: lauv-noptilus-1
```

Figure B.1: Extract of Dolphin runtime log

Appendix C

Generated data from vehicles logs

Vehicles individual timelines obtained in Neptus Mission Review and Analysis feature based on the log files downloaded from the vehicles.

C.1 Rendezvous scenario at APDL

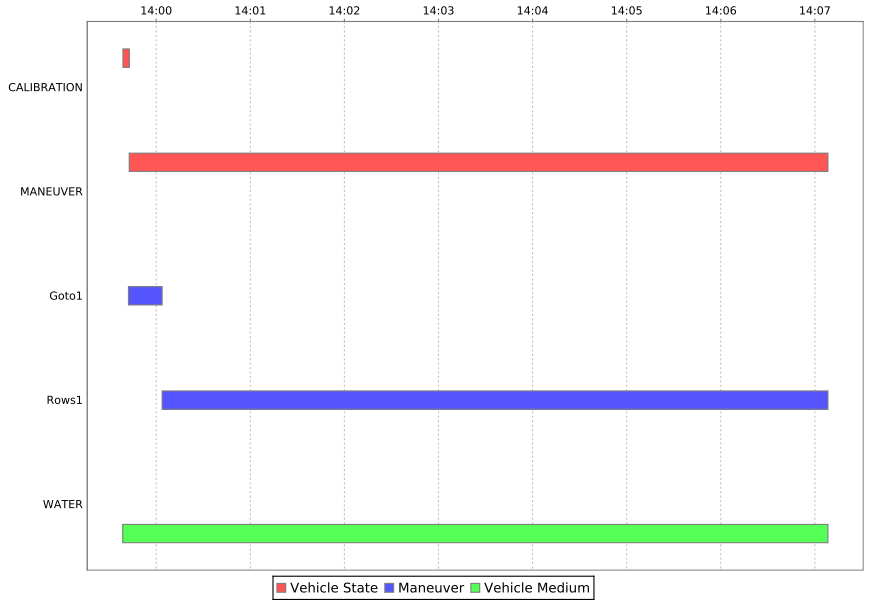
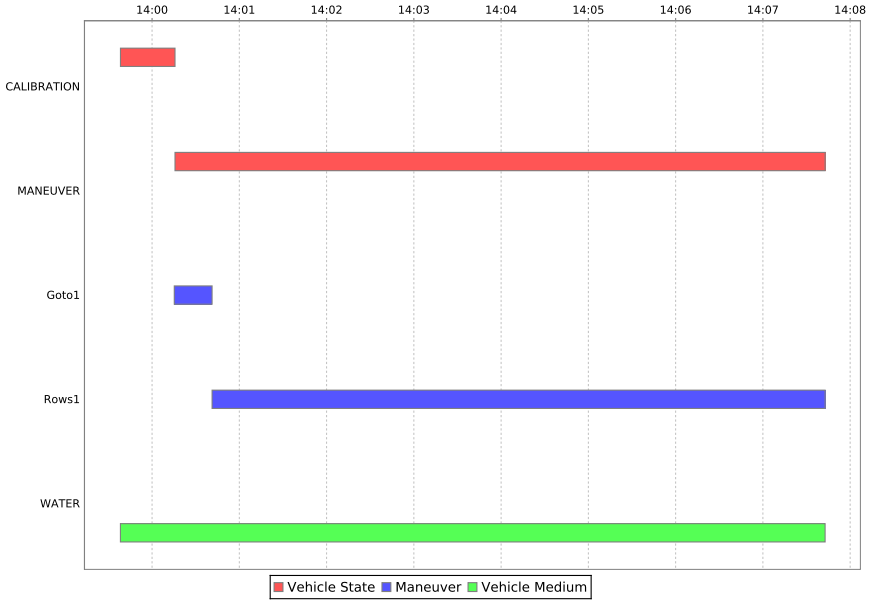
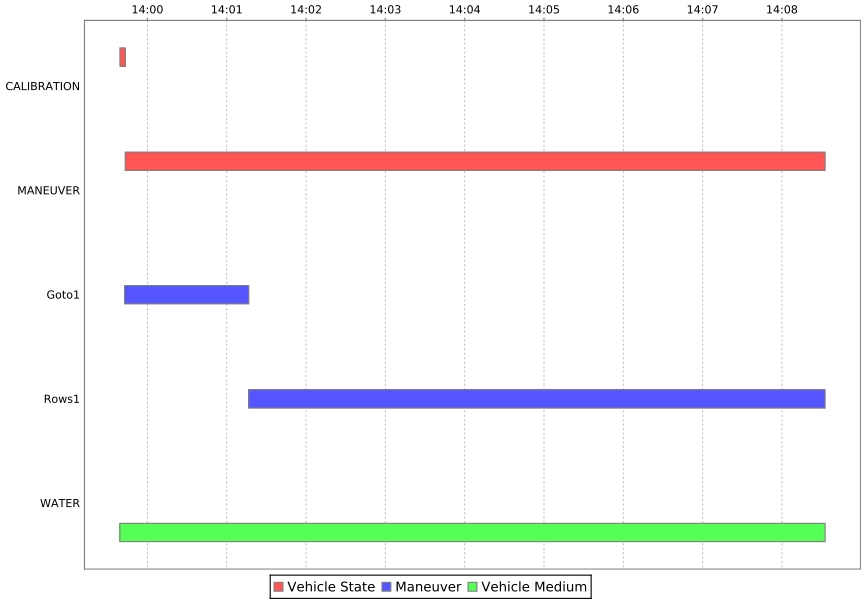


Figure C.1: LAUV-Noptilus-1 execution timeline



(a) LAUV-Noptilus-2 execution timeline

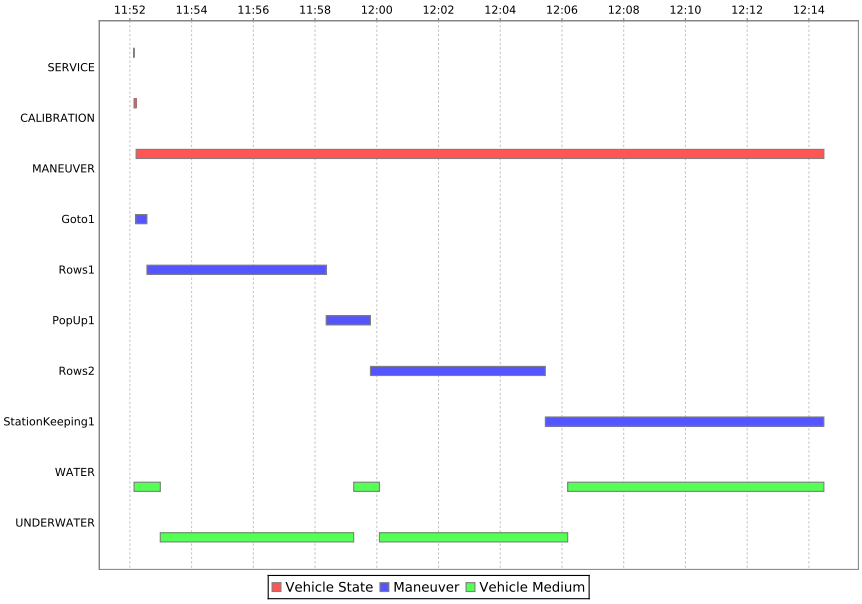


(b) LAUV-Noptilus-3 execution timeline

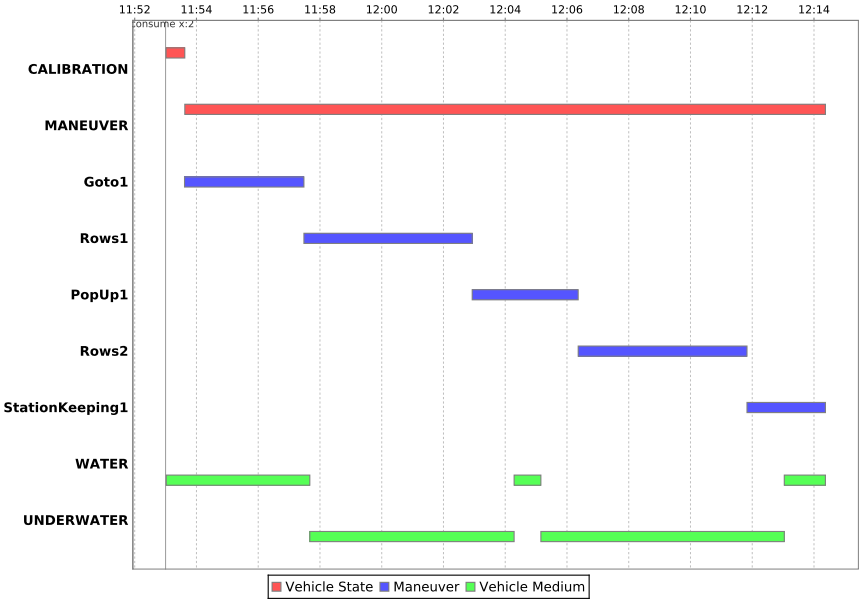
Figure C.2: APDL field tests UUVs timelines.

C.1.1 Execution Synchronization using Events

Figure C.3: REP'17 day 1 - Events and synchronization LAUV-Xplore-1/master timeline



(a) LAUV-Noptilus-1 execution timeline



(b) LAUV-Noptilus-2 execution timeline

Figure C.4: REP'17 day 1 - Events and synchronization slaves timelines

C.2 Rendezvous scenario reviewed at REP17

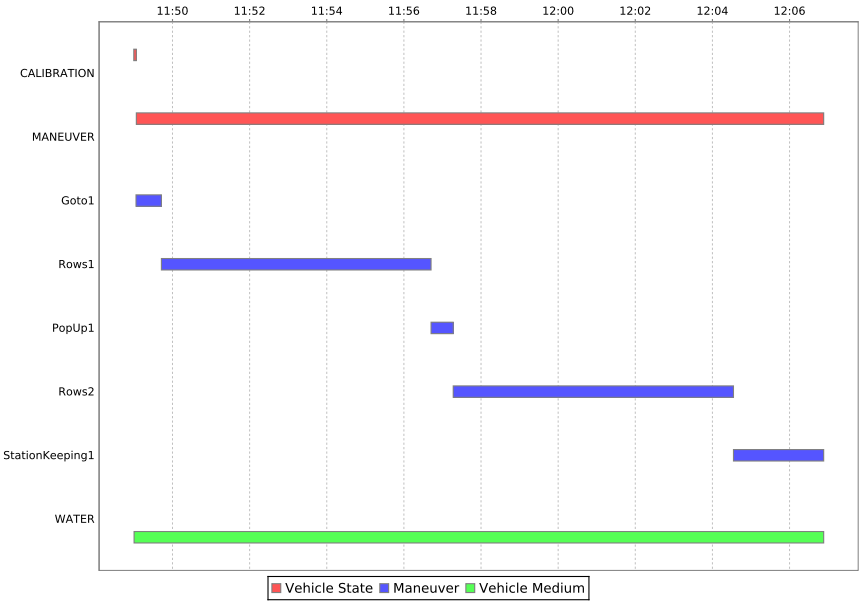
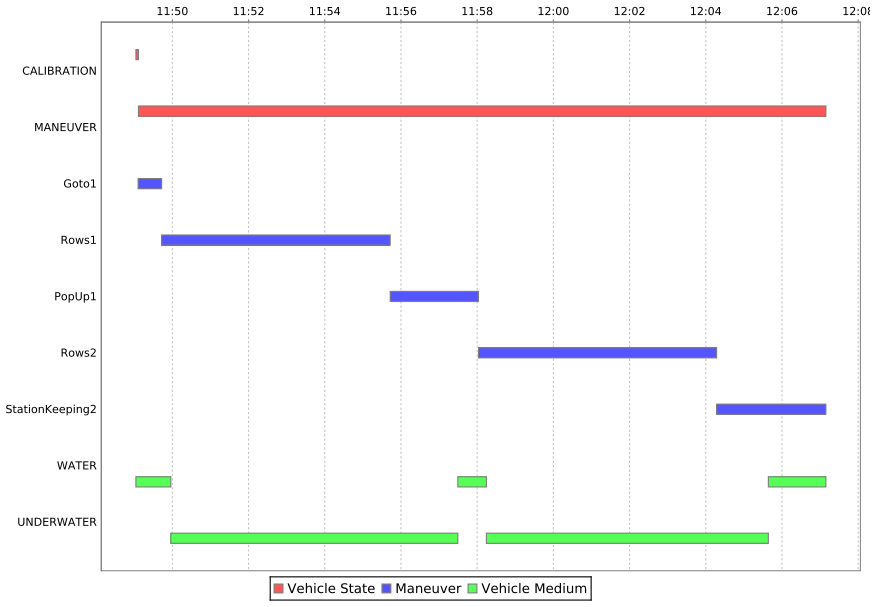
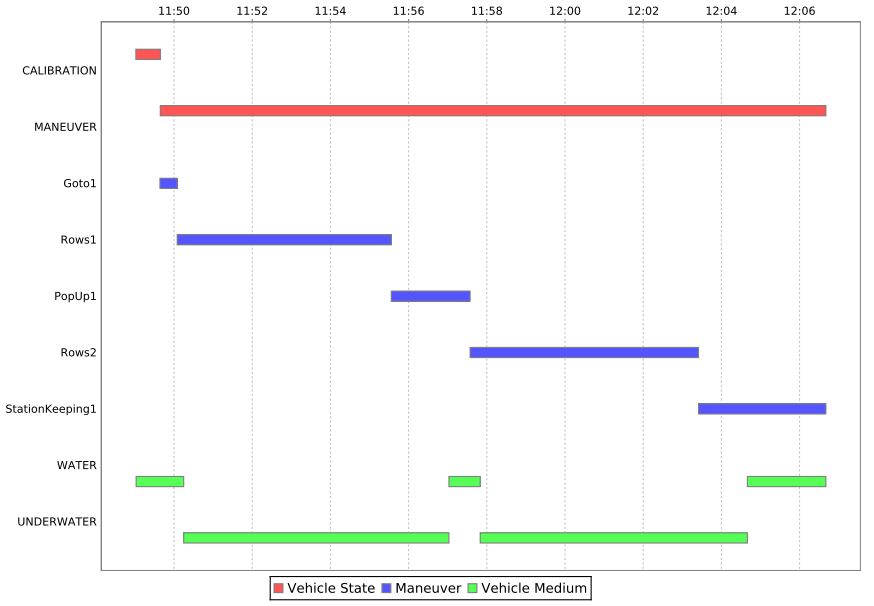


Figure C.5: REP'17 day 2 - Rendez-vous scenario LAUV-Xplore-1 execution timeline



(a) LAUV-Noptilus-1 execution timeline



(b) LAUV-Noptilus-2 execution timeline

Figure C.6: REP'17 day 2 - Rendez-vous scenario timelines

Appendix D

Groovy Plug-in

Scripts used at vehicles recovery and individual vehicles timelines from Neptus[1]

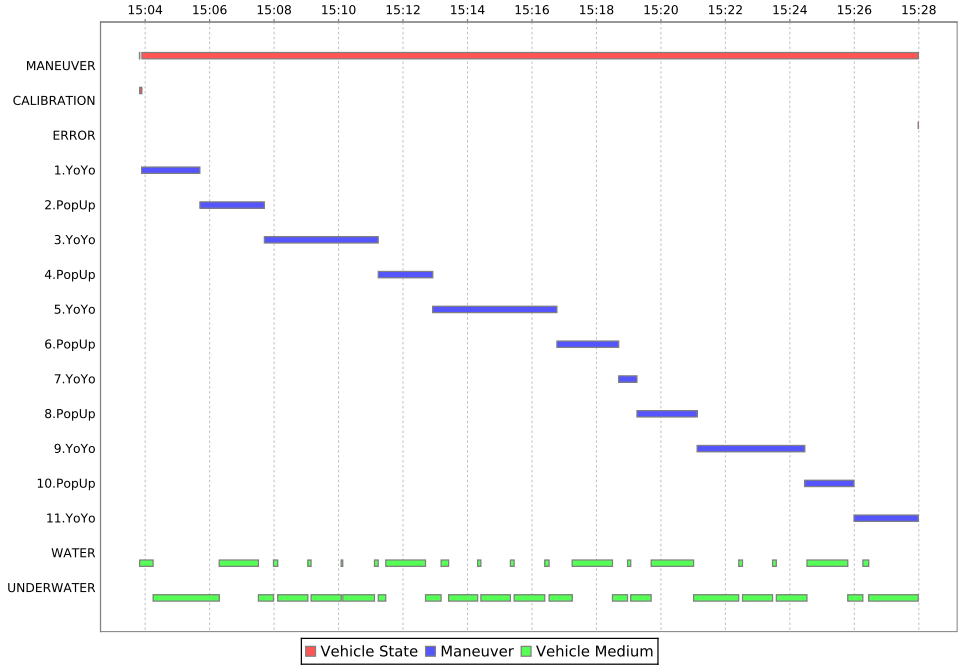


Figure D.1: LAUV-Xplore-2 Yoyo-Popup script execution timeline at DRiP mission

[1]The timelines were generated from vehicles downloaded logs.

```
1  //-----Parameters Definition-----
2
3  def popDistance    = 500 //in meters
4  def plan           = "plan1"   // already defined in the console
5  double speed_value = 1350.0
6  def speed_unit     = "RPM"
7  double z_value     = 0.0
8  def z_unit         = "DEPTH"
9  if(plans.get(plan)==null)
10         println "Could not find initial plan "+plan
11 def initialPlan = new Plan(console,plans.get(plan))
12 def start = initialPlan.initialLocation()
13 def waypoints = initialPlan.waypoints()
14
15 new Plan(console).with{
16
17         planName "YoYo_enter_river"
18         def startPoint = start
19         speed speed_value,speed_unit
20         z z_value,z_unit
21    //each waypoint of the original plan
22         waypoints[0..waypoints.size-1].each(){
23                 waypointYoYo ->
24                 midpoints(startPoint,waypointYoYo,popDistance).each{
25                         yoyo location:it,max_depth:6.0,min_depth:0.0
26                         popup(waitAtSurface:true,duration:60)
27                         }
28                 startPoint = waypointYoYo
29                 }
30
31             vehicles "lauv-xplore-1"
32             addToConsole()
33 }
```

Figure D.2: Vehicle recovery - First attempt script

```
def loc1 = locations.get 'final'
def start = locations.get 'xp1'
def p1 =new Plan(console)

p1.with {
        planName "final_point"
        speed 1500,"RPM"
        yoyo location: start,max_depth:8, min_depth:0
        yoyo location: loc1, max_depth:8, min_depth:0
        midpoints (start,loc1,400).each{
                popup(location:it,duration:60)
                }
        addToConsole()
}
```

Figure D.3: Vehicle recovery - Second attempt script

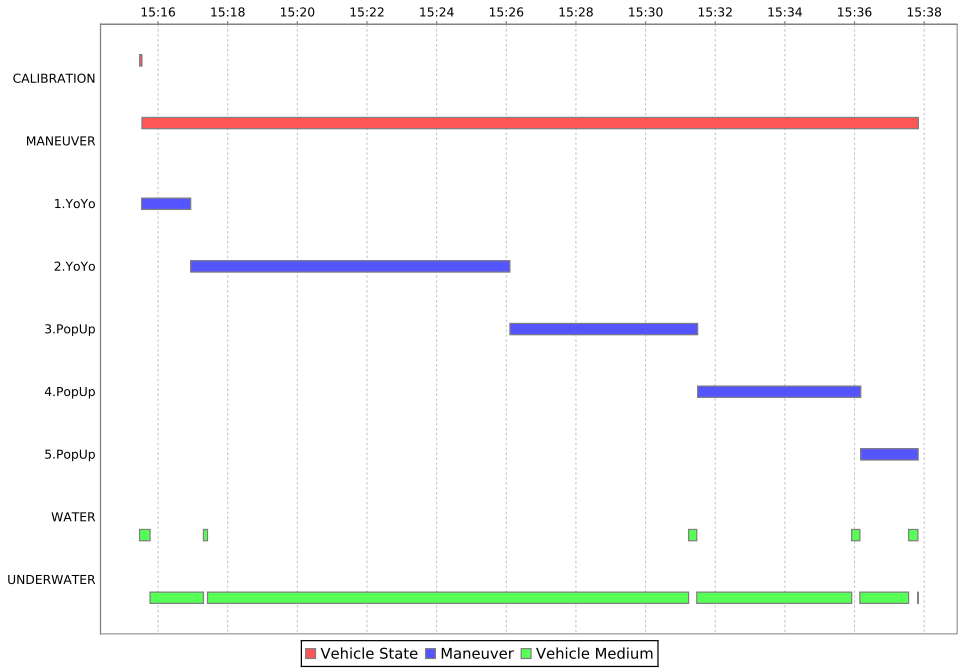


Figure D.4: LAUV-Xplore-1 execution timeline at DRiP mission.