Programming Networked Vehicle Systems using Dolphin – Field Tests at REP'17

Keila Lima*, Eduardo R. B. Marques†, José Pinto*, João B. Sousa*
* Laboratório de Sistemas e Tecnologia Subaquática, Faculdade de Engenharia, Universidade do Porto, Portugal
Email: {keila,zepinto}@lsts.pt, jtasso@fe.up.pt
†CRACS/INESC-TEC & Faculdade de Ciências da Universidade do Porto, Portugal
Email: edrdo@dcc.fc.up.pt

***Abstract*—The increasing availability and use of autonomous vehicles for real operational scenarios has led to the need for tools that allow human operators to interact with multiple systems effectively, taking into account their capabilities, limitations and environmental constraints. Multiple vehicles, deployed together in order to accomplish a common goal, impose a high burden on a human operator for specifying and executing coordinated behavior, particularly in mixed-initiative systems where humans are part of the control loop. In this paper, we describe experimental field tests for Dolphin, a domain-specific language that allows a single program to define the joint behaviour of multiple vehicles over a network. Using the language, it is possible to accomplish an orchestrated execution of single-vehicle tasks according to several patterns such as sequential, concurrent, or event-based program flow. With this aim, Dolphin has been integrated modularly with a software toolchain for autonomous vehicles developed by Laboratório de Sistemas e Tecnologia Subaquática (LSTS). The tests we describe made use of LSTS unmanned underwater vehicles (UUVs) at open sea during the 2017 edition of Rapid Environment Picture (REP), an annual exercise jointly organised by LSTS and the Portuguese Navy.**

I. Introduction

The use of autonomous vehicles is now massive for a variety of scientific, military, and civilian applications. Moreover, quite often and increasingly, several vehicles are deployed together in a dynamic networked environment that typically also includes human operators and external sensors in the control loop. The resulting networked vehicle systems presents manifold challenges, arising from the multiple layers of complexity and dynamics inherent to such a system-of-systems. For sound and automated operation, a lower burden on the human users, and more advanced multi-vehicle applications, there is a pressing need for convenient models that treat a networked vehicle system as a whole in terms of intent and behavior, and which are effectively materialised in practice by software tools. More specifically, we are concerned with top-level programmability of networked vehicle systems, i.e., the question of how to to program the coordinated behavior of components in a networked vehicle system based on a top-level specification.

With this general problem in mind, we recently developed Dolphin [1, 2], a programming language for autonomous vehicle networks available open-source[1]. A Dolphin program expresses an orchestrated execution of tasks that are defined for multiple vehicles dynamically available in a network. The built-in Dolphin operators include support for composing one-vehicle tasks running in different vehicles, for instance according to concurrent, sequential, or event-based program flow. Dolphin is embedded as a domain-specific language (DSL) in Groovy [3], allowing a direct interplay with Groovy/Java software packages and simple addition of DSL extensions. A Dolphin runtime has been developed for the the open-source LSTS toolchain for autonomous vehicles [4]. The toolchain provides support for the execution of single-vehicle tasks, called IMC plans, instantiating the elementary unit of computation within Dolphin programs.

Using Dolphin, we were able to orchestrate IMC plans in expressive manner for multiple vehicles, as demonstrated by field test trials. This paper describes the use of Dolphin at the 2017 edition of the Rapid Environment Picture (REP)[2]. REP is an annual exercise organised by LSTS and the Portuguese Navy, with the overall aim of testing the combined use of autonomous vehicles, navy vessels and personnel. Each edition of REP takes several weeks and involves several invited partners and assorted scientific and technological goals, e.g., see [5, 6] for a description of REP'16 and REP'15. In 2017, REP'17[3] lasted for two weeks, and the field tests for Dolphin were conducted for two days.

The rest of the paper is structured as follows. Section II gives an overview of Dolphin and its integration with the LSTS toolchain, and the example programs we tested during REP'17. Section III presents the field tests results. Section IV concludes the paper with an outlook of future work.

II. Overview of Dolphin

A. Architecture

The use of Dolphin in integration with the LSTS toolchain is schematically depicted in Fig. 1. As illustrated, the idea is that a user is capable of programming a network of vehicles using Dolphin programs. The three LSTS toolchain modules [4] at stake comprise: (1) IMC, a message-based protocol for interoperability between nodes in the network such as vehicles or human operator consoles; (2) Neptus, a command-and-control infrastructure that allows users to prepare, monitor, and

[1] http://DolphinDSL.github.io

[2] http://rep.lsts.pt

[3] http://rep17.lsts.pt

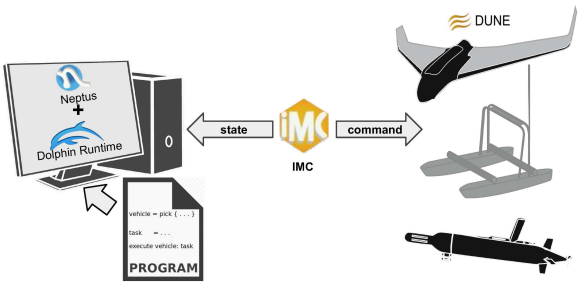


Fig. 1. Integration of Dolphin with the LSTS toolchain.

review vehicle operations using GUI consoles, and; (3) DUNE: an on-board software platform for autonomous vehicles.

Dolphin defines a set of abstract programming bindings that must be instantiated through plug-ins for concrete software platform of interest [1] like the LSTS toolchain. The primary aspects of instantiation comprise networked interactions (network node characterisation, discovery, etc) and the nature of single-vehicle tasks. In the case of the LSTS toolchain, these aspects are instantiated through the IMC protocol. In particular, an IMC message subset is dedicated to single-vehicle tasks, called IMC plans, that can be specified using Neptus by a human user and executed onboard a vehicle using DUNE. An IMC plan is a sequence of maneuvers for a single vehicle, where each maneuver may express relatively simple behaviours such as waypoint tracking or more complex ones such as area surveys.

The integration of Dolphin with the LSTS toolchain takes form through a Neptus plugin for editing and running Dolphin programs, depicted in Fig. 2, or, alternatively, a standalone command-line tool. The Dolphin plugin for Neptus embeds the Dolphin runtime in the overall Neptus GUI, providing human operators a friendly environment for editing and executing Dolphin programs along with associated IMC plans. In complement, a special-purpose DSL for IMC plans [1] also allows a programmatic definition of IMC plans within Dolphin code. No changes were required to IMC or DUNE otherwise

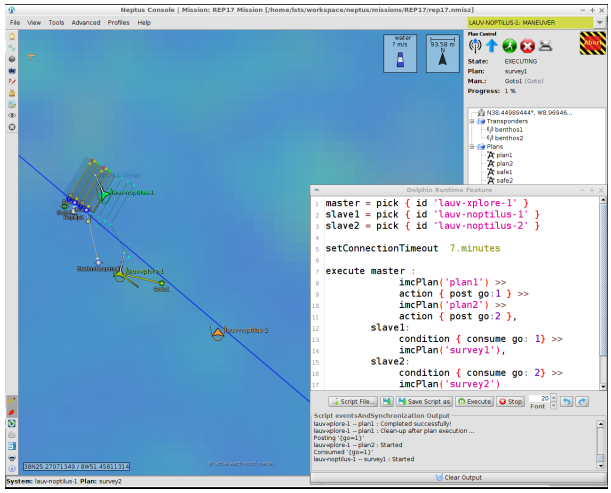
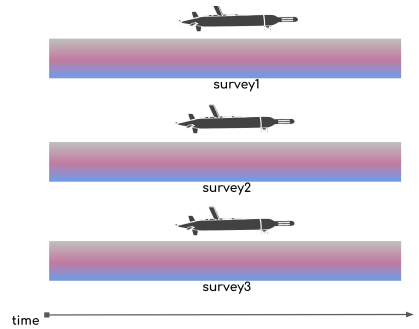


Fig. 2. Dolphin plugin running in Neptus.

to support the execution of Dolphin programs. Hence, the integration of Dolphin with the toolchain is a modular one, a strategy we intend to pursue for other autonomous vehicle platforms, e.g., ongoing work targets unmanned aerial vehicles that communicate using the MAVLink protocol[4].

B. Example programs

We now provide an overview of some of Dolphin's features through two example programs that correspond to the field test results we present later in Section III.



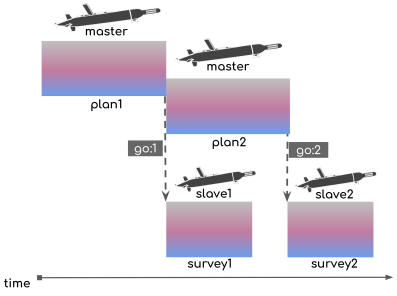

```
1  // Define area of operation.
2  Comporta = (location 38.43462, -8.86118) ^ 2.km
3
4  // Select UUVs.
5  UUVs = pick {
6           count 3
7           type 'UUV'
8           region Comporta
9           payload 'Sidescan'
10        }
11
12 // Set connection timeout.
13 setConnectionTimeout 10.minutes
14
15 // Execute ocean surveys.
16 execute UUVs:
17           imcPlan['survey1'] |
18           imcPlan['survey2'] |
19           imcPlan['survey3']
20
21 // End!
22 message 'Done!'
```

Fig. 3. Example Dolphin program – concurrent ocean surveys.

The first example program, shown in Fig. 3, illustrates a simple pattern of concurrent task execution. Three UUVs are first selected from the network using the **pick** instruction (line 5), each situated in a region identified as **Comporta** and equipped with a side-scan sonar. The selection output variable, **UUVs**, represents the set of all three vehicles. After selection, and before any actual operation with the vehicles, a connection timeout is set to 10 minutes (line 13), given that the UUVs may be underwater for a significant amount of time, and will in that case only communicate state in sparse intervals through limited means (e.g., underwater acoustic modems). Actual execution is specified by the following **execute** instruction (line 5), that tasks the vehicles with three IMC plans, corresponding to ocean surveys in separate areas, to be executed concurrently (one per vehicle), as specified by the use of the "|" Dolphin operator. To match vehicles and

[4]http://mavlink.io/en; check http://DolphinDSL.github.io for further information about MAVLink integration with Dolphin.

IMC plans, the Dolphin engine allocates each vehicle in the **UUVs** set to a requested IMC plan employing an heuristic that tries to minimize the distance of each vehicle's current position to the start location of an IMC plan. After all the surveys are complete, the program ends with a simple output message (line 22).




```
1  // Select UUVs.
2  master = pick { id 'lauv-xplore-1' }
3  slave1 = pick { id 'lauv-noptilus-1' }
4  slave2 = pick { id 'lauv-noptilus-2' }
5
6  // Set connection timeout.
7  setConnectionTimeout  7.minutes
8
9  // Execute tasks.
10 execute master :
11              imcPlan('plan1') >>
12              action { post go:1 } >>
13              imcPlan('plan2') >>
14              action { post go:2 },
15         slave1:
16              condition { consume go: 1} >>
17              imcPlan('survey1'),
18         slave2:
19              condition { consume go: 2} >>
20              imcPlan('survey2')
21
22 // End!
23 message 'Done!'
```

Fig. 4. Example Dolphin program – master/slave scenario.

The second Dolphin program, shown in Fig. 4, illustrates the use of other Dolphin features regarding vehicle selection and task composition operators. The overall scenario again involves 3 UUVs, but with two distinct roles: one UUV acts as "master" for task coordination while the other two UUVs act as "slaves", in the sense that the slaves' operation is dependent on events related to the flow of the master. More specifically, the event flow is such that the completion of IMC plans by the master is required before plans at each of the slaves may execute.

In the first example, all three vehicles were selected "anonymously" and treated as a group. In this case, each UUV is selected separately and identified precisely by their identifier (lines 2–4; note that a **count 1** default configuration is implicit in each **pick** instruction). Three singleton sets identify each vehicle: **master**, **slave1**, and **slave2**. The task flow specified by the subsequent **execute** instruction is also quite different (line 10). The concurrent behavior is specified separately per UUV, each in terms of a sequential task composition, as dictated by the use of the "**>>**" operator. Each slave UUV waits for a notification, **go:1** for **slave1** and **go:2** for **slave2**, before engaging in a survey plan, **survey1** and **survey2** respectively. The notifications are delivered in line with the master UUV's execution: **go:1** is delivered after the master completes **plan1**, at which point **slave1** may proceed with **survey1** and; similarly, **go:2** is delivered after the master completes **plan2**, letting **slave2** proceed with **survey2**. Note that we can attain different task flows by rearranging the order of the the master UUV's sequential actions, without needing to change the specification for the slave UUVs. For instance, if we change the program such that **go:1** and **go:2** are posted only after all master UUV plans complete, then the surveys at the slave UUVs would start simultaneously and only after the master UUV was idle.

III. Field Tests

We now present the results for the field tests we conducted during the REP'17 exercise. We first describe the overall field test setup and then present the results for the two Dolphin programs introduced in the previous section.

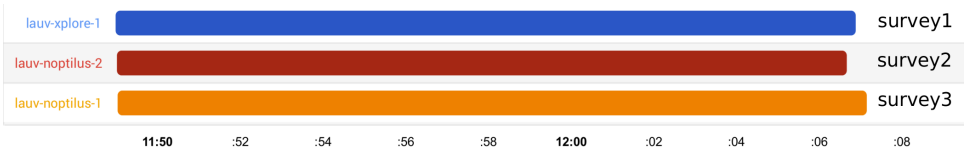
(a) Test location.

(b) NRP Cassiopeia and autonomous vehicles.

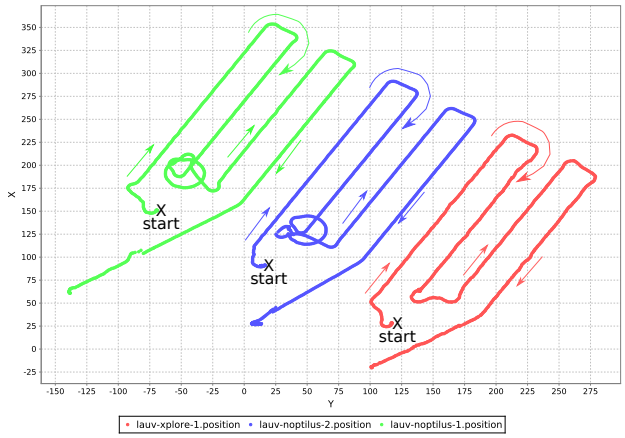
Fig. 5. Field test setup.

A. Setup

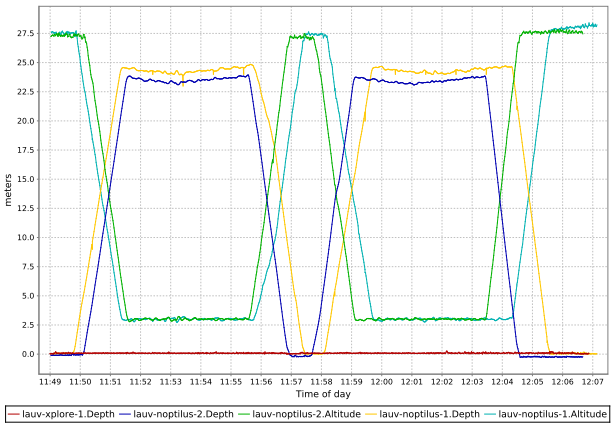
REP'17 took place at the Tróia peninsula, Portugal, during two weeks in July. The Dolphin tests were conducted on July 10/11, 2017, near Comporta beach, as depicted by the map in Fig. 5a that includes an overlay identifying the exact area of operation. Human operators, together with Navy personnel, were onboard the NRP Cassiopeia vessel, depicted in the photo of Fig. 5b along with some of the autonomous vehicles used during REP'17.



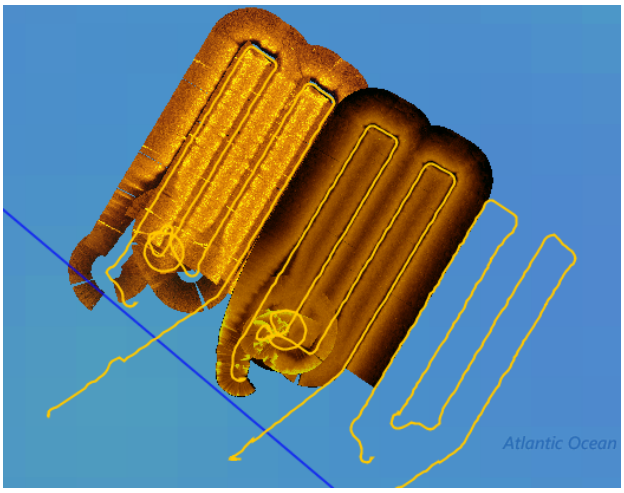
(a) Execution timeline.



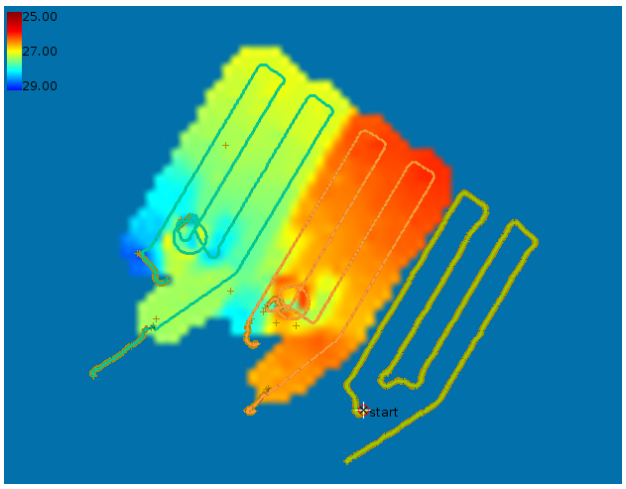
(b) UUV positions.



(c) UUV altitude/depth.



(d) Sidescan sonar data.



(e) Bathymetry data.

Fig. 6. Results for the ocean surveys program.

The UUVs we used were three LAUV-class vehicles [7], designed to operate continuously for up to 12 hours travelling at 3 knots, with two distinct payload configurations: *Noptilus1* and *Noptilus2* are vehicles equipped with DVL and side scan sonars, making them them suitable for bathymetry or sea-floor mapping, and; *Xplore1* is a vehicle equipped with an environmental probe measuring temperature, salinity, pH and Redox, making it suitable for water column surveys. For homogeneous operation, the third vehicle we had in mind was *Noptilus3*, a vehicle bearing the same characteristics as *Noptilus1* and *Noptilus2*, but logistical issues involving that vehicle implied that we had to resort to *Xplore1* instead.

For communication between operator consoles and vehicles, we used a Manta communications gateway [4] that provided WiFi and interface with underwater acoustic modems in a range of approximately 1 kilometre. The main state relevant to Dolphin programs could only be obtained through WiFi, since data transmitted over acoustic modems did not convey full information regarding IMC plan execution. Acoustic communications nevertheless provided useful feedback in the form of periodical updates for vehicle positions.

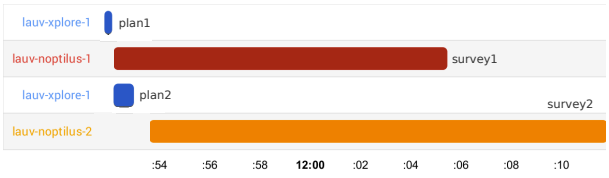
B. Results

We exercised the ocean surveys scenario/program of Fig. 3 as follows. First, given that *Xplore1* was not equipped with a side scan, we adjusted the Dolphin program not to require that payload for the vehicles, i.e., we removed the `payload 'SideScan'` constraint (at line 9 in Fig. 3). The IMC survey plans were designed to conduct a global survey split in different areas by the 3 UUVs. Each plan included two "row" maneuvers, that make a vehicle cover rectangular areas in legs, mediated by a vehicle "pop-up", i.e., a maneuver that makes a UUV come at the surface and remain there for a

specified amount of time. A pop-up may be useful for several purposes, e.g., to adjust navigation errors or to communicate data via Wifi or Iridium. From the Dolphin program's perspective, the purpose of the pop-ups was having check-points to ensure the vehicles were executing the expected IMC plans.

The results of the ocean surveys program are shown in Fig. 6. The main observations are as follows:

- The execution timeline (6a) shows that all three surveys executed concurrently and took roughly 17 minutes to conclude. This duration is more than the connectivity timeout set in the Dolphin program (10 minutes), but the pop-up maneuvers we configured for the IMC plans (making the vehicle surface in-between rows and communicate via WiFi) prevented timeout expiration.
- The UUV positions' plot (6b) for vehicle positions illustrates the two row maneuvers mediated by a pop-up per each survey plan/vehicle.
- The altitude/depth plot indicates that the surveys for *Noptilus1* and *Noptilus2* were set up for seafloor tracking, parameterised by a distance of 3 meters to the sea-floor, while the survey for *Xplore1* was set up for the surface.
- The side scan (6d) and bathymetry (6e) plots show that we were able to obtain a mapping of the sea floor in regard to the areas in which *Noptilus1* and *Noptilus2* operated. *Xplore1* did not have sensors to measure altitude and, as mentioned earlier, was not also equipped with a side scan sonar.



(a) Execution timeline

(b) UUV positions.

Fig. 7. Results for the master/slave program.

Let us now turn to the master-slave program/scenario (of Fig. 4) and the corresponding results shown in Fig. 7. Recall that the Dolphin code (lines 2–4 in the program) specifies a master role for *Xplore1* and slave roles for *Noptilus1* and *Noptilus2*. The slave plans (**`survey1`** and **`survey2`**) were similar in nature to the ocean surveys scenario, and the master plans (**`plan1`** and **`plan2`**) consisted of simple waypoint tracking patterns over a short area, thus taking a relatively short amount of time; our aim was merely testing event-based coordination in Dolphin in simple manner. The results are depicted in terms of an execution timeline (7a) and UUV vehicle positions (7b). The timeline illustrates that the desired orchestration of IMC plans was fulfilled, i.e., the master executes **`plan1`** and **`plan2`** in sequence, and **`survey1`** and **`survey2`** initiate at distinct times defined by the completion of each of the master's plans. The UUV position plot depicts the slave vehicles' surveys along with the short paths for the master's plans (shown bottom-right). We omit altitude/depth data, plus bathymetry and side scan plots derived from *Noptilus1* / *Noptilus2* data, that have a similar nature to the oceans survey scenario.

IV. Conclusion

We presented the use of the Dolphin language in field tests involving UUVs deployed at ocean sea during the REP'17 exercise. The type of scenarios we considered are relatively simple, for example in comparison to the survey/rendez-vous scenario we describe in [1] involving more complex language features and distinct types of vehicles, but also what we have in mind for language developments and field tests in the future.

Some key conceptual and technical requirements will drive subsequent work, such as: supporting cooperative vehicle tasks that may also allocate vehicles dynamically, as opposed to the current static and one-to-one allocation of tasks to vehicles that is also unable to adapt to vehicle churn; language constructs for a a richer characterisation of task and event flow, e.g., to express patterns of persistent operation over time and space, or to account for varying network conditions, and; finally, also the need to interface with vehicles enabled by software platforms other than the LSTS toolchain.

Acknowledgments

This work was partially funded by European Commission, H2020 and DG ECHO, under the projects BRIDGES (GA 635359) and E-UReady4OS (GA 740129) and by ERDF (P2020), under projects Marinfo (NORTE-01-0145-FEDER-000031) and SMILES/TEC4GROWTH (NORTE-01-0145-FEDER-000020).

References

[1] K. Lima, E. R. B. Marques, J. Pinto, and J. B. Sousa, "Dolphin: a task orchestration language for autonomous vehicle networks," in *arXiv:1803.00944 – submitted to IROS'18*, 2018.

[2] K. Lima, "Dolphin: A domain-specific language for autonomous vehicle networks," Master's thesis, MIERSI/DCC/FCUP, 2017.

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

[4] J. Pinto, P. S. Dias, R. Martins, *et al.*, "The LSTS toolchain for networked vehicle systems," in *Proc. Oceans'13*. IEEE, 2013.

[5] A. S. Ferreira, J. Pereira, J. Pinto, *et al.*, "Rapid environmental picture atlantic exercise 2016: Field report," in *Proc. Oceans'17*. IEEE, 2017.

[6] J. B. de Sousa, J. Pereira, J. Pinto, *et al.*, "Rapid environmental picture atlantic exercise 2015: A field report," in *Proc. Oceans'16*. IEEE, 2016.

[7] L. Madureira, A. Sousa, J. Braga, *et al.*, "The Light Autonomous Underwater Vehicle: Evolutions and networking," in *Proc. Oceans'13*. IEEE, 2013.