NVL: a coordination language for unmanned vehicle networks

Eduardo R. B. Marques[1], Manuel Ribeiro[2], José Pinto[2], João B. Sousa[2], Francisco Martins[1]

[1] LASIGE/Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa

[2] Laboratório de Sistemas e Tecnologia Subaquática, Faculdade de Engenharia, Universidade do Porto

ABSTRACT

The coordinated use of multiple unmanned vehicles over a network can be employed for numerous real-world applications. However, multi-vehicle operations are often deployed through a patchwork of separate components that informally "glue" together during operation, as they are hard to program as a "whole". With this aim, we developed the Networked Vehicles' Language (NVL) for coordinated control of unmanned vehicle networks. A single NVL program expresses an on-the-fly selection of multiple vehicles and their allocation to cooperative tasks, subject to time, precedence, and concurrency constraints. We present the language through an example application involving unmanned underwater vehicles (UUVs) and unmanned aerial vehicles (UAVs), the core design and implementation traits, and results from simulation and field test experiments.

1. INTRODUCTION

Unmanned vehicles are nowadays routinely employed for numerous applications, many of which can benefit from the coordinated behaviour of multiple vehicles over a network [1, 6]. Moreover, large systems are now being deployed with a massive integration of unmanned vehicles, sensors, and human user interaction, that can also be spatially distributed across the globe, e.g., [11, 15] in the realm of oceanography. It is generally hard to program the software components in these systems as a "whole", so that they work cooperatively towards a common goal. In particular, coordinated vehicle operations often need to be carefully scripted through human intervention, with low automation and informality in what concerns the specification of a "network program" that implements a scenario of interest. The path from modelling abstractions (e.g., [17, 18]) to their realisation, i.e., actual software programs, is still in its infancy.

In this context, we introduce the Networked Vehicle's Language (NVL), a language for coordination of unmanned vehicle networks. The idea is that a single NVL program specifies the coordination of a global scenario involving multiple vehicles. At the basis of NVL, we consider a resource-task model. A vehicle is seen as a resource that is capable of accomplishing tasks on its own or cooperatively with other vehicles. NVL takes the view that the resource (vehicle) set changes over time, due to constraints of autonomy, mobility, or connectivity in vehicle operation. As such, NVL programs select the required vehicles for task execution on-the-fly from the "network cloud". A task, in turn, is modelled as an indivisible unit of timed computation that requires one or more vehicles in order to execute. Tasks can be composed in a sequential or concurrent manner through a base primitive that instantiates the popular fork-join programming model, with appropriate refinements for the definition of timing constraints and explicit resource awareness. The language is complemented with imperative programming constructs for program control flow.

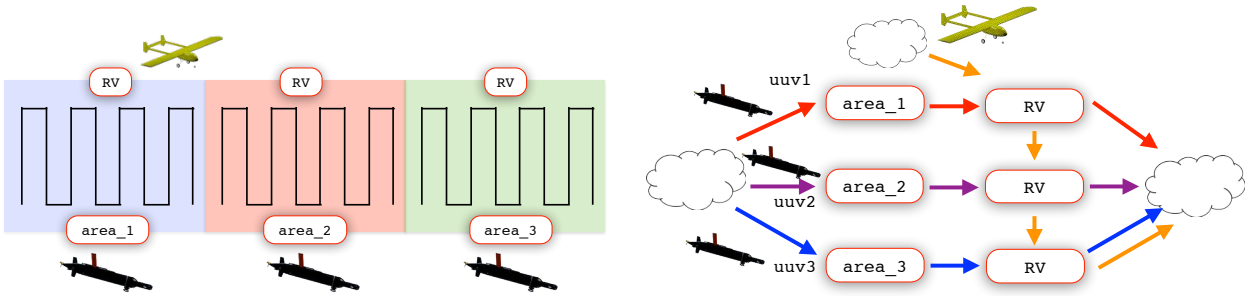
Related work. There are several heterogeneous approaches for modelling the coordination of tasks in unmanned vehicle networks, e.g., dynamic and hierarchical hybrid automata networks [18], "vignette scripts" that map onto abstract state machines [17], the use of distributed deliberative planning using "timelines" [16], combined bigraph/actor models [14], or dynamically-changeable Petri nets [12]. The concerns of NVL are largely orthogonal to these works. The language focuses on the low-level foundational programming constructs that can be used as core building blocks for coordinating unmanned vehicle networks. Our aim is that NVL can either be used directly by a programmer, or as a backend language by higher-level modelling frameworks or semantic abstractions. In programming language terms, NVL borrows the fork-join programming model found in task-based concurrent programming [3, 10]. The NVL constructs are also timed, an essential trait of coordination languages for distributed cyber-physical systems [7, 9].

Paper outline. The rest of the paper provides an informal overview of NVL. We present the language using an example scenario (Section 2) involving unmanned underwater vehicles (UUVs) and unmanned aerial vehicles (UAVs), the core design and implementation traits (Section 3), and results from simulation environments and field tests with real vehicles (Section 4). We end the paper with a discussion of future work (Section 5).

2. THE NVL LANGUAGE

Example scenario. We begin by describing an example scenario, depicted in Fig. 1. Overall, common patterns of spatial task decomposition and data muling in the operation of unmanned vehicles are at stake [6, 8]. Three UUVs are used for environmental data sampling (e.g., bathymetry measurements) across a given ocean region, together with an UAV that acts as a "data mule" to collect the data from the UUVs. As illustrated in Fig. 1 (a), the sampling region is distributed evenly between the three UUVs, defining distinct areas for operation and corresponding sampling tasks that take form as "row pattern" maneuvers (area_1, area_2, and area_3). The UUVs

SAC'15 , April 13-17 2015, Salamanca, Spain.
 ACM 978-1-4503-3196-8/15/04 ...$15.00
http://dx.doi.org/10.1145/2695664.2696029



(a) Spatial distribution of tasks (b) Task flow

Figure 1: Example scenario

emerge after data sampling and the UAV collects their data through three instances of a cooperative rendezvous task (RV), involving the UAV and each of the UUVs. A UAV-UUV rendezvous makes the UAV move close to the UUV operation area for improved connectivity first, after which the actual data transfer proceeds.

The task execution flow is illustrated in Fig. 1 (b). The UUV sampling tasks execute concurrently, and once they are over,the rendezvous tasks execute in order between UAV and each UUV. The "clouds" in the figure indicate that vehicles are selected on-the-fly from the network cloud, rather than being defined a priori, and, in symmetry, that a vehicle is released back to the cloud as soon as its job is finished.

The NVL program. The NVL program for the example scenario is shown in Fig. 2. A program consists of a set of task declarations and a set of procedures. The example comprises four tasks (lines 2–6) and two procedures (8–39). We consider some possible refinements of the program later in this section, but in any case the code shown illustrates the core traits of the language.

```
// Task declarations                                          1
task area_1 (vehicle uuv);                                    2
task area_2 (vehicle uuv);                                    3
task area_3 (vehicle uuv);                                    4
task RV (vehicle uav, vehicle uuv)                            5
  yields done, moreData;                                      6
// Main procedure                                             7
proc main() {                                                 8
  // Select UUVs.                                             9
  select 5 m {                                                10
    uuv1 uuv2 uuv3 (type: "UUV")                              11
  } then {                                                    12
    // Execute sampling tasks.                                13
    step 60 m {                                               14
      area_1(uuv1)                                            15
      area_2(uuv2)                                            16
      area_3(uuv3)                                            17
    }                                                         18
    // Select UAV.                                            19
    select 5 m {                                              20
      uav (type: "UAV")                                       21
    } then {                                                  22
      // Execute rendezvous tasks.                            23
      call rendezvous(uav, uuv1)                              24
      call rendezvous(uav, uuv2)                              25
      call rendezvous(uav, uuv3)                              26
      message "done"                                          27
    }                                                         28
  }                                                           29
}                                                             30
// Rendezvous execution                                       31
proc rendezvous(vehicle uav, vehicle uuv) {                   32
  do {                                                        33
    step 15 m {                                               34
      rvRes = RV(uav, uuv)                                    35
    }                                                         36
  }                                                           37
  while (rvRes = moreData)                                    38
}                                                             39
```

Figure 2: NVL program for the example scenario

Tasks are declared with a name, a set of required vehicles, and optional completion results; they are called cooperative if they require more than one vehicle. In the example, the UUV sampling tasks (area_1, area_2, and area_3, lines 2–4) require just one vehicle (uuv) and declare no completion results. The cooperative rendezvous task (RV, 5–6) requires two vehicles (uav and uuv) and declares two completion results (moreData and done). The idea is that a rendezvous may complete by signalling either an incomplete but resumable data transmission (moreData) or full completion (done).

NVL procedures are sequences of instructions that express on-the-fly selection of vehicles and subsequent allocation of selected vehicles to tasks. The execution of a program starts with procedure main by convention. In the example, main (lines 8–30) works as follows:

— The procedure begins with a **select** instruction (line 10) to acquire the UUVs for data sampling (uuv1, uuv2, and uuv3), with a selection deadline of 5 minutes. The program stops if (any of) the vehicles cannot be selected within this time, in line with the default error handling mechanism discussed later in this section.

— If the UUVs are successfully selected, the **then** block associated to the **select** instruction is carried out (lines 12–29). The **then** block begins by executing the three UUV data sampling tasks concurrently through a **step** instruction (14–18), with a shared deadline of 60 minutes. The program will not advance until all tasks in the step body (area_1, area_2, and area_3) complete. A UUV may thus remain idle (i.e., executing some fallback behavior such as loitering) after it completes its task, waiting for other UUV tasks to also complete.

— After the data sampling step, a UAV is selected (line 20) to execute the rendezvous tasks cooperatively with the UUVs, through calls to procedure rendezvous (24–26). This procedure (32–39) defines a **do-while** loop that executes the RV task until data transmission is completed.

Fig. 3 depicts a possible execution for the program, identifying time intervals for the execution of **select** and **step** instructions. Observe that, for a **step** instruction, network constraints may lead to some delay in realising that tasks did complete by the NVL execution engine. In the worst case, these delays may lead to failure

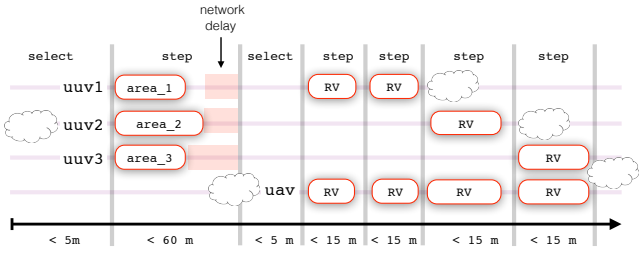


Figure 3: Execution of the NVL program

in the execution of a **step** instruction. Regarding the rendezvous stage, the figure illustrates that the rendezvous procedure may fire the RV task more than once (shown for uuv1/uav).

Vehicle selection criteria. In the example program, the class of vehicles is the single selection criterium: type: "UUV" (Fig. 2, line 11) and type: "UAV" (21). Thus, any UUVs and UAV available in the network can be selected by the program. NVL supports complementary selection criteria to attend to specific requirements of spatial locality, vehicle payload, or a non-anonymous choice of vehicle. For instance,

```
select ... {
  uav (type: "UAV", id: "eagle_1",
       area: (41.1830, -8.7000, 1.0),
       hasPayload: "Camera" )
} then { ... }
```

tries to select an UAV named eagle_1, located within a 1 km range of latitude-longitude coordinates 41.1830 N 8.7 W, and with an onboard camera as part of the payload.

Error handling. NVL supports **or** instruction blocks that are executed in case of failure during the execution of **select** and **step**. The general syntax is as follows: **select** ... **then** ... **or** { <instr>∗ } and **step** ... { ... } **or** { <instr>∗ }. An **or** { **exit** } block is assumed as default for both **select** and **step**. The **exit** instruction releases all previously selected vehicles by the program from duty and ends execution. Since the example program defined no **or** blocks, it will then end execution if any of the **select** or **step** instructions fail. To handle an error due to the first **select** in the example program, an **or** block could be used:

```
select ...
then { ... }
or {
  delay 1h
  continue main
}
```

In the variation above, the **or** block employs a **delay** instruction to pause execution for one hour, and a **continue** instruction to subsequently restart executing the main procedure.

3. IMPLEMENTATION

Architecture. The architecture of the current NVL implementation is depicted in Fig. 4. Specific to NVL, there are three software components: an integrated development environment (IDE) for writing and validating NVL programs, a language interpreter that executes programs, and NVL supervisors that run onboard unmanned vehicles on behalf of program execution. The architecture also employs three other components from the open-source software toolchain for unmanned vehicles developed at Laboratório de Sistemas e Tecnologia Subaquática (LSTS), described in [16] and available from http://github.com/LSTS: DUNE is the system used to program the onboard software of unmanned vehicles; Neptus is a command-and-control system for human operators to configure, plan, and monitor unmanned vehicles using GUI consoles; and IMC is an extensible message-based protocol for networked interoperability.

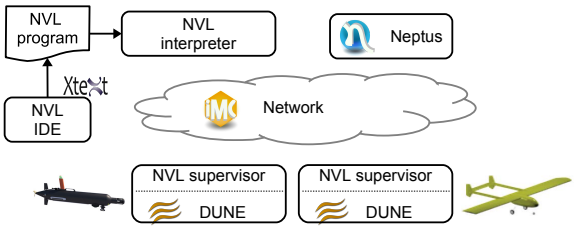


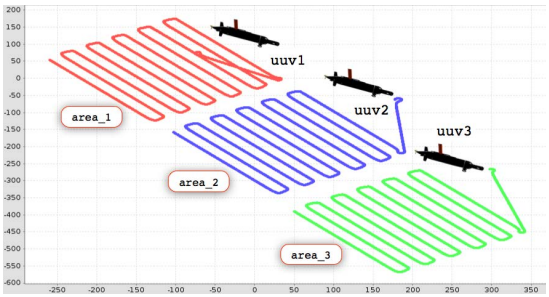
Figure 4: Implementation architecture

NVL IDE. The NVL IDE is employed by a user to write and validate a program within the popular Eclipse environment for software development. The tool is developed using Xtext [2], an open-source toolkit for implementing domain-specific languages. Xtext provides convenient support for typical tasks in language design and implementation, such as the definition of the language grammar or semantic validation.

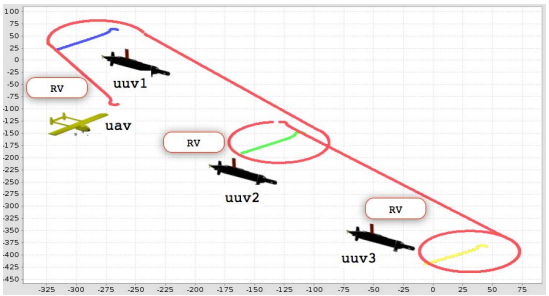
Execution environment. The execution of a program comprises the interaction between the interpreter and the supervisors onboard each NVL-enabled vehicle. The code of both these components is written in Java, and executes using the low-footprint Java SE Embedded runtime environment. Supervisors attend to the interpreter's orders for task execution, and report back related state, mediating access to the local DUNE instance that directly controls the vehicle. Supervisors also launch controllers for tasks (that run within the supervisor) to interface with that DUNE instance. When cooperative tasks are at stake (like the rendezvous in our running example), controllers in distinct vehicles also interact among themselves through supervisor-to-supervisor communication. The Neptus system is used to design maneuver plans triggered by task controllers and to monitor the progress of these plans during execution. All communication between components in this environment takes form through IMC messages transmitted over the network.

4. EXPERIMENTS

Simulation of the example program. To simulate the execution of NVL programs, the software architecture described previously can be configured to use physical simulation engines within DUNE in place of actual vehicles [16]. Using this setup, we ran our example program at the simulated physical location of the Leixões harbour near Porto, where we also conducted subsequent field tests described below. Data sampling tasks were defined by row-pattern maneuver plans designed using Neptus. These plans were fired directly by NVL supervisors to DUNE instances. For the rendezvous tasks, we also employed maneuver plans that were fired by rendezvous controllers running within the NVL supervisors. Fig. 5



(a) Data sampling stage



(b) Rendezvous stage

Figure 5: Example scenario – simulation plots

shows plots of the resulting execution, comprising the data sampling stage by the three UUVs (a) and subsequent rendezvous with the UAV (b). The simulation took 45 minutes to complete, split between 29 minutes for the data sampling tasks and 16 minutes for the rendezvous tasks.

Field tests. We conducted preliminary field tests at the Leixões harbour in a single day of September 2014. We had two LAUV-Seacon [13] UUVs (Fig. 6) from LSTS available. Given the tight schedule, there was only time to reproduce the UUV sampling step plus individual rendezvous steps from the example program, along with a series of prior sanity checks. The NVL interpreter ran on a laptop computer, and the NVL supervisors ran onboard the UUV vehicles using Beagle Bone devices equipped with a 1 GHz ARM CPU and 512 MB of RAM. For the data sampling step, we employed a third simulated UUV, running on another laptop computer together with the corresponding NVL supervisor. The data sampling took 20 minutes, quite less than in the simulation setup (29 minutes), as we had to constrain the operation area for harbour security reasons, but both simulation and field test executions were otherwise similar. For the individual rendezvous steps we used the two (real) UUVs, with one of the UUVs acting as the data mule.

Figure 6: LAUV-Seacon vehicle

5. FUTURE WORK

We now discuss a few key items for future work.

Regarding the NVL resource-task model and associated semantics, there is a need for more flexibility and expressiveness. In a variation of the example scenario, for instance, one could wish for the UAV to opportunistically engage in rendezvous with each UUV, and doing so in any order as each terminates their data sampling operation without waiting for the other UUVs. In contrast, the example program waits for completion of the data sampling step in all UUVs, before executing the three rendezvous in a strict order. For this goal, refined synchronisation constructs from fork-join based programming languages can be particularly relevant [3, 4, 10]. We are also interested in extensions for supporting vehicle "teams" and associated dynamics [17, 18].

In a second line of work, NVL programs can potentially be analysed formally in respect to timing constraints, resource (vehicle) requirements and allocation, and environmental constraints. The following general problem of feasibility may be stated: can program P accomplish its tasks in t time with a vehicle set V under constraints C ? In relation, we are interested in establishing design contracts [5] for programs and their formal verification.

Finally, we are considering a tighter integration with the LSTS toolchain, including the development of a Neptus NVL plugin, and using NVL as a backend language for deliberative planning [16]. For these purposes, the language also needs to contemplate aspects like fine-grained error handling, human operator inputs, and more programmer-friendly language constructs.

6. REFERENCES

[1] J. Bellingham and K. Rajan. Robotics in remote and hostile environments. *Science*, 318(5853):1098–1102, 2007.

[2] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2014.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proc. OOPSLA*. ACM, 2005.

[4] T. Cogumbreiro, F. Martins, and V. T. Vasconcelos. Coordinating phased activities while maintaining progress. In *Proc. COORDINATION*. Springer, 2013.

[5] P. Derler, E. Lee, S. Tripakis, and M. Törngren. Cyber-physical system design contracts. In *Proc. ICCCPS*. ACM, 2013.

[6] M. Dunbabin and L. Marques. Robots for environmental monitoring: Significant advancements and applications. *IEEE Robotics Automation Magazine*, 19(1):24–39, 2012.

[7] J. Eidson, E. Lee, S. Matic, S. Seshia, and J. Zou. Distributed Real-Time Software for Cyber–Physical Systems. *Proc. IEEE*, 100(1):45–59, 2012.

[8] M. Faria, J. Pinto, F. Py, J. Fortuna, H. Dias, R. Martins, F. Leira, T. Johansen, J. Sousa, and K. Rajan. Coordinating UAVs and AUVs for Oceanographic Field Experiments: Challenges and Lessons Learned. In *Proc. ICRA*, 2014.

[9] T. Henzinger, C. Kirsch, E. Marques, and A. Sokolova. Distributed, modular HTL. In *Proc. RTSS*. IEEE CS, 2009.

[10] S. Imam and V. Sarkar. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proc. PPPJ*. ACM, 2014.

[11] A. Isern and H. Clark. The Ocean Observatories Initiative: A continued presence for interactive ocean research. *Marine Technology Society Journal*, 37(3):26–41, 2003.

[12] J. Love, J. Jariyasunant, E. Pereira, M. Zennaro, K. Hedrick, C. Kirsch, and R. Sengupta. CSL: A Language to Specify and Re-Specify Mobile Sensor Network Behaviors. In *Proc. RTAS*. IEEE, 2014.

[13] L. Madureira, A. Sousa, J. Braga, P. Calado, P. Dias, R. Martins, J. Pinto, and J. Sousa. The Light Autonomous Underwater Vehicle: Evolutions and networking. In *Proc. Oceans*. IEEE, 2013.

[14] E. Pereira, C. Kirsch, R.Sengupta, and J. Sousa. BigActors - A Model for Structure-aware Computation. In *Proc. ICCPS*. ACM, 2013.

[15] C. Petrioli, R. Petroccia, J. Potter, and D. Spaccini. The SUNSET framework for simulation, emulation and at-sea testing of underwater wireless sensor networks. *Ad Hoc Networks and Physical Communication*, 2014.

[16] J. Pinto, P. Dias, R. Martins, J. Fortuna, E. Marques, and J. Sousa. The LSTS Toolchain for Networked Vehicle Systems. In *Proc. Oceans*. IEEE, 2013.

[17] H. Shahir, U. Glässer, R. Farahbod, P. Jackson, and H. Wehn. Generating test cases for marine safety and security scenarios: a composition framework. *Security Informatics*, 1(1):1–21, 2012.

[18] J. Sousa, T. Simsek, and P. Varaiya. Task planning and execution for UAV teams. In *Proc. CDC*. IEEE, 2004.