COLA2: A Control Architecture for AUVs

Narcis Palomeras, Andres El-Fakdi, Marc Carreras, and Pere Ridao

***Abstract*—This paper presents a control architecture for an autonomous underwater vehicle (AUV) named the Component Oriented Layer-based Architecture for Autonomy (COLA2). The proposal implements a component-oriented layer-based control architecture structured in three layers: the reactive layer, the execution layer, and the mission layer. Concerning the reactive layer, to improve the vehicle primitives' adaptability to unknown changing environments, reinforcement learning (RL) techniques have been programmed. Starting from a learned-in-simulation policy, the RL-based primitive *cableTracking* has been trained to follow an underwater cable in a real experiment inside a water tank using the *Ictineu* AUV. The execution layer implements a discrete event system (DES) based on Petri nets (PNs). PNs have been used to safely model the primitives' execution flow by means of Petri net building block (PNBBs) that have been designed according to some reachability properties showing that it is possible to compose them preserving these qualities. The mission layer describes the mission phases using a high-level mission control language (MCL), which is automatically compiled into a PN. The MCL presents agreeable properties of simplicity and structured programming. MCL can be used to describe offline imperative missions or to describe planning operators, in charge of solving a particular phase of a mission. If planning operators are defined, an onboard planner will be able to sequence them to achieve the proposed goals. The whole architecture has been validated in a cable tracking mission divided in two main phases. First, the *cableTracking* primitive of the reactive layer has been trained to follow a cable in a water tank with the *Ictineu* AUV, one of the research platforms available in the Computer Vision and Robotics Group (VICOROB), University of Girona, Girona, Spain. Second, the whole architecture has been proved in a realistic simulation of a whole cable tracking mission.**

***Index Terms*—Mission programming, Petri nets (PNs), reinforcement learning (RL), robot control architectures, underwater vehicles.**

I. Introduction

OVER the past few years a considerable interest has arisen around autonomous underwater vehicle (AUV) applications. The capabilities of such robots as well as their mission requirements have been increased. Industries from around the world call for technology applied to several underwater scenarios, such as environmental monitoring, oceanographic research, or maintenance/monitoring of underwater structures. AUVs are attractive for utilization in these areas but their comparison with human beings in terms of efficiency and flexibility still favors remotely operated vehicles (ROVs) in most cases. The development of autonomous control systems able to deal with these issues becomes a priority. The use of AUVs for covering large unknown underwater areas is a very complex problem. Although technological advances in marine vehicles offer highly reliable devices, control architectures have the complex and unsolved task to adapt the AUV behavior in real time to the unpredictable changes in the environment.

A control architecture [1] is the part of the robot control system which makes the decisions. An autonomous robot is designed to accomplish a particular mission, and the control architecture must achieve this mission by generating the proper actions. As shown in [2], several questions should be answered before implementing a robotic architecture: Which kind of mission the robot is designed for? Which set of actions are needed to perform this mission? Which data are necessary to do it? Who are the robot users? How will the robot be evaluated? An organized analysis of every question allows a successful implementation of a particular architecture. Starting from the kind of mission to be performed by current AUVs, they range from surveys to inspections, passing through intervention operations or target location missions. To achieve these missions, AUVs generally have a set of primitives which describe simple tasks that the vehicle is able to perform autonomously like "go to a point," "follow a pipeline," or "reach a desired depth." The data needed by these primitives are acquired by a complete sensor suite. To improve the robustness of the gathered data, auxiliary modules are also designed to fuse, filter, and refine it. Unlike many applications for mobile robots like surveillance [3], mail delivering [4], or tour guides [5], where the robot has some previously defined goals and it decides, under some constraints, how to achieve them, AUV users are mainly scientists or engineers who program their vehicles with a certain degree of autonomy sufficient to generalize a mission. Therefore, on the one hand, robots in underwater domains do not generally decide "where to go" or "what to do"; instead, in most commercial systems, underwater vehicles usually follow a predefined mission plan. However, on the other hand, as the knowledge about underwater environments is often uncertain or incomplete, modern underwater vehicle control architectures must offer a certain degree of deliberation.

Flexibility and adaptation is one of the primary goals in a control architecture. During a mission, underwater robots are immersed in a dynamic and changing world model and equipped with sensors that can be notoriously imprecise and noisy [6]. AUVs must be able to change their behavior at any given time. For this purpose, recent control architectures aim for different methodologies to adapt to new incoming events that may occur during a planned mission. In a control architecture, adaptation

Manuscript received January 27, 2011; revised May 14, 2012; accepted June 13, 2012. Date of publication August 08, 2012; date of current version October 09, 2012. This work was supported by the TRIDENT EU FP7-Project under Grant ICT-248497 and the Spanish Government under Project DPI2008-06548-C03-03.

Guest Editor: M. Seto.

The authors are with the Computer Vision and Robotics Group (VICOROB), Computer Engineering Department, University of Girona, 17071 Girona, Spain (e-mail: npalomer@eia.udg.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JOE.2012.2205638

capabilities may be implemented at different levels. A high-level deliberative layer can modify a predefined mission plan if a sensor failure occurs, batteries are too low, or external conditions do not allow the whole fulfillment of the initial mission. Also, from the low-level point of view, the performance of a control architecture can be improved by developing better primitives which constitute the action–decision methodology. The success of a control architecture directly depends on the success of its primitives. Therefore, different techniques such as reinforcement learning (RL), genetic algorithms, or adaptive control, can modify the parameterization of the different primitives for a better performance of the whole execution process.

The research presented in this paper develops the Component Oriented Layer-based Architecture for Autonomy (COLA2), an architecture specially designed for AUVs. This architecture represents an improvement from the previously developed Object Oriented Control Architecture for Autonomy (O2CA2) [7]. As a novelty, mission plans are defined and executed by means of Petri nets (PNs) [8]. These plans can be computed by an onboard planner or compiled from a high-level description given by a user. PNs are naturally oriented toward the analysis and modeling of discrete event systems (DESs) allowing the user to control the flow between the tasks to be executed to carry out a particular mission. The proposed control architecture introduces also the utilization of RL [9] techniques in the reactive layer. RL automatically interacts with the environment finding the best mapping for the proposed robot primitive. The architecture is implemented on *Ictineu* AUV [10] and tested in the context of an underwater cable inspection mission, combining experiments performed in a water tank with simulated results.

This work is organized as follows. Initially, a brief overview of control architectures for autonomous robots is given in Section II. We first review the history of control architectures for autonomous robots, starting with traditional methods of artificial intelligence (AI) and ending with the widely used layer-based architectures. The section ends describing the control architecture proposed in this paper. Section III describes the reactive layer together with the RL-based algorithm used to program the cable tracking primitive. Sections IV and V describe the execution and mission layers, respectively. The experimental application where the proposed architecture is used to perform an autonomous cable tracking mission is presented in Section VI. Finally, results are given in Section VII before concluding this paper in Section VIII.

II. An Overview of Control Architectures

The first attempt at building autonomous robots began around the mid-20th century with the emergence of AI. The approach begun at that time was known as traditional AI, classical AI, or deliberative approach. Traditional AI relied on a centralized world model for verifying sensory information and generating actions in the world model, following the sense, plan, and act (SPA) pattern [11]. Its main architectural features were that sensing flowed into a world model, which was then used by the planner, and that plan was executed without directly using the sensors that created the model. The design of the classical control architecture was based on a top–down philosophy. The sequence of phases usually found in a traditional deliberative control architecture can be seen in Fig. 1. Real robot architectures using a SPA deliberative control architecture began in the late 1960s with the Shakey robot at Stanford University (Stanford, CA) [12], [13]. Many other robotic systems have been built with the traditional AI approach [14]–[18], all of them sharing the same kind of problems. Planning algorithms failed with nontrivial solutions and the integration of the world representations was extremely difficult and, as a result, planning in a real-world domain took a long time. Also, using the sensors only during the world sensing step and not during the plan execution is dangerous in a dynamic world model. Only structured and highly predictable environments were proved to be suitable for classical approaches.

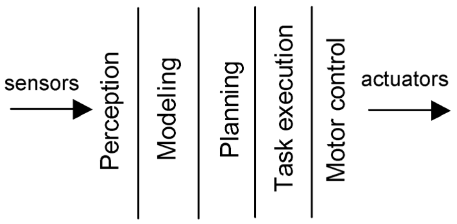


Fig. 1. Phases of a classical deliberative control architecture.

In the middle of the 1980s, due to dissatisfaction with the performance of robots in dealing with the real world, a number of scientists began rethinking the general problem of organizing intelligence. Among the most important opponents to the AI approach were Brooks [19], Rosenschein and Kaelbling [20], and Agre and Chapman [21]. They criticized the symbolic world which traditional AI used and wanted a more reactive approach with a strong relation between the perceived world and the actions. They implemented these ideas using a network of simple computational elements, which connected sensors to actuators in a distributed manner. There were no central models of the world explicitly represented. The model of the world was the real one as perceived by the sensors at each moment. Leading the new paradigm, Brooks proposed a subsumption architecture. The subsumption architecture is built from layers of interacting finite-state machines (FSMs). These FSMs were called behaviors, representing the first approach to a new field called behavior-based robotics. The behavior-based approach used a set of simple parallel behaviors which reacted to the perceived environment proposing the response the robot must take to accomplish the behavior (see Fig. 2). Whereas SPA robots were slow and tedious, behavior-based systems were fast and reactive. There were no problems with world modeling or real-time processing because they constantly sensed the world and reacted to it. Successful robot applications were built using the behavior-based approach, most of them at the Massachusetts Institute of Technology (MIT, Cambridge, MA) [22], [23]. A well-known example of behavior-based robotics is Arkin's motor-control schema [24], where motor and perceptual schema are dynamically connected to one another.

Despite the success of the behavior-based models, they soon reached limits in their capabilities. Limitations when trying to undertake long-range missions and the difficulty to optimize the robot behavior were the most important difficulties encountered. Also, since multiple behaviors can be active simultaneously, behavior-based architectures need an arbitration mech-

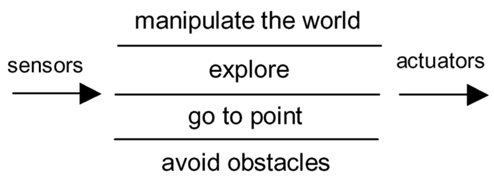


Fig. 2. Structure of a behavior-based control architecture.

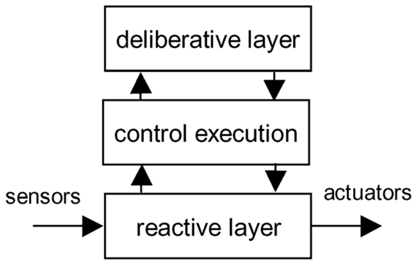


Fig. 3. The hybrid control architecture structure.

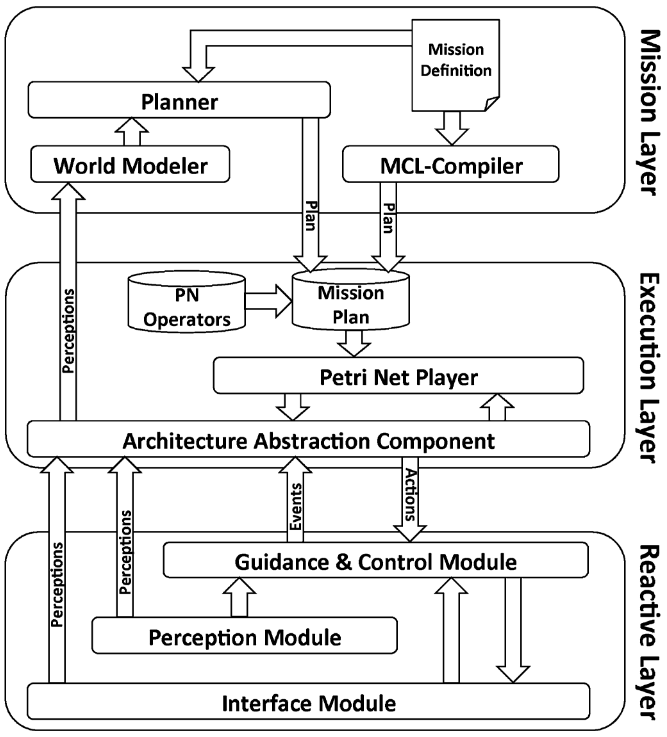


Fig. 4. Three-layer organization control architecture.

anism to either enable higher level behaviors to override signals from lower level behaviors or to blend multiple behaviors responses into a single one. Therefore, another difficulty has to be solved: How do we select the proper behaviors for robustness and efficiency in accomplishing goals? In essence, robots needed to combine the planning capabilities of the classical architectures with the reactivity of the behavior-based architectures, attempting a compromise between bottom–up and top–down methodologies. This evolution was named layered architectures or hybrid architectures. As a result, most of today's architectures for robotics follow a hybrid pattern. Usually, a hybrid control architecture is structured in three layers: the reactive layer, the control execution layer, and the deliberative layer (see Fig. 3). The reactive layer takes care of the real-time issues related to the interactions with the environment. It is composed of the robot primitives where the necessary sensors and/or actuators are directly connected. Each primitive can be designed using different techniques, ranging from optimal control to RL techniques. The control execution layer interacts between the upper and lower layers, supervising the accomplishment of the tasks. This layer acts as an interface between the numerical reactive layer and the symbolical deliberative layer interpreting high-level plan primitives' activation. The control execution layer also monitors the primitives being executed and handles the events that these primitives may generate. The deliberative layer transforms the mission into a set of tasks which define a plan. It determines the long-range tasks of the robot based on high-level goals. One of the first researchers who combined reactivity and deliberation in a three-layered architecture was Firby [25]. Since then, hybrid architectures have been widely used. One of the best known is Arkin's autonomous robot architecture (AURA) [26], where a navigation planner and a plan sequencer were added to its initial behavior-based motor-control schema architecture. The planner–reactor architecture [27] and the Atlantis [28] used in the Sojourner Mars explorer are well-known examples of hybrid architectures as well.

The architecture proposed in this paper implements a layered architecture model but with some peculiarities: RL [9] techniques are used to improve the primitive adaptation in the reactive layer; PNs [8] are used to describe the mission plans to be executed by the vehicle; and, finally, task planning techniques are added to increase the deliberation capabilities of the whole system.

A. Our Proposal

A preliminary version of the presented architecture COLA2 was first developed for the *Ictineu* AUV [10], a robot designed to take part in the 2006 Student Autonomous Underwater Challenge—Europe (SAUC-E). A new prototype, named *Sparus* AUV, which has participated in the 2010 and 2011 SAUC-E editions, awarded with the first and second places, respectively, also implements the same architecture. The COLA2 is a component oriented architecture where each component may exist autonomously from other components in a computer node. Components have the ability to communicate with each other even if they are in a different computer. All the components in the architecture are distributed in three hierarchical layers. Despite the facilities provided by popular frameworks, libraries, and middlewares like Robot operating system (ROS) [29], Player & Stage [30], or Mission Orientated Operating Suite (MOOS) [31], a custom framework has been developed to simplify the implementation of each component. This framework provides the necessary elements to communicate the components among them through shared memory or Transmission Control Protocol/Internet Protocol (TCP/IP), save logs, control the execution of multiple threads, and control the access to external devices as serial ports or cameras.

Although the separate use of RL techniques, PNs, and planning algorithms is not new, their combination into an AUV control architecture is a novelty. Following the layer-based model, the components in the control architecture are distributed among

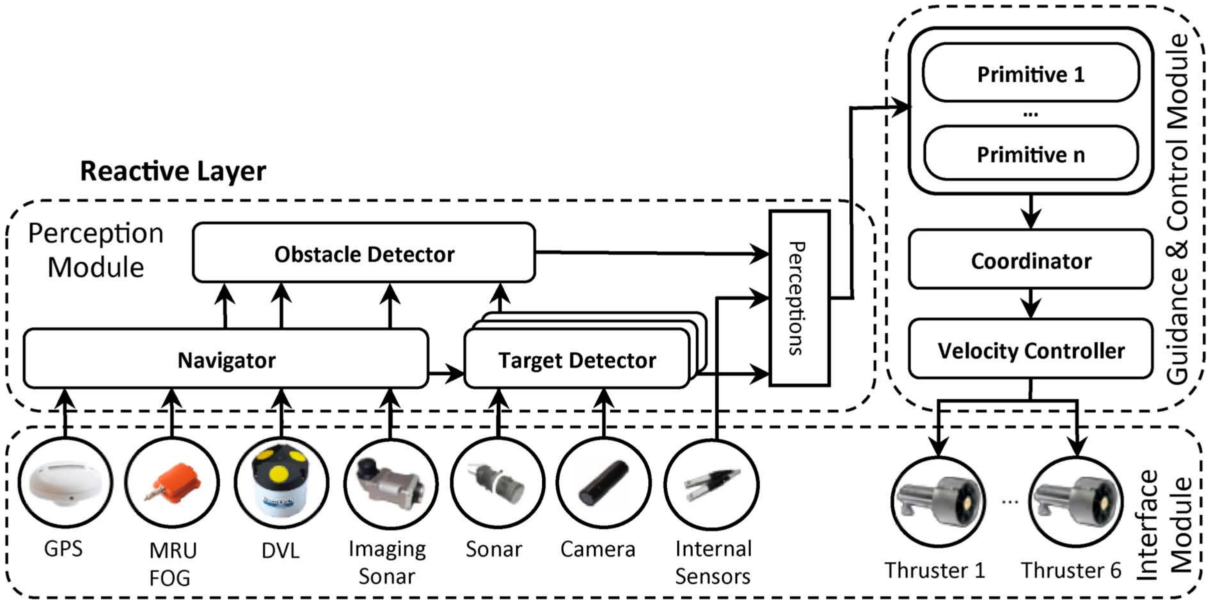


Fig. 5. The COLA2 reactive layer.

three hierarchical layers, as shown in Fig. 4. The mission definition/deliberation or simply mission layer, the control execution or simply execution layer, and the reactive layer compose the proposed architecture. The mission layer obtains a mission plan by means of an onboard automatic planning algorithm or compiling a high-level mission description given by a human operator. The mission plan is described using a PN that is interpreted by the execution layer. This plan is executed by enabling/disabling available vehicle primitives contained in the reactive layer. Since the adaptation of the main primitives to the environment is crucial for the mission success, RL methods are used to improve them. A description of the three layers is given next.

III. The Reactive Layer

The reactive layer executes basic primitives to fulfill the mission plan given by the mission layer. Primitives (also known as behaviors in other layer-based architectures) are basic robot functionalities offered by the robot control architecture. For an AUV, a primitive can range from a basic sensor enabling [e.g., start a camera sensor *camera(enable)*] to a complex primitive activation [e.g., navigate towards a 3-D way point *goto(x, y, z)*].

Although available sensors and actuators will affect mission capabilities and planning, the mission and execution layers can be considered almost vehicle independent; on the other hand, the reactive layer is very dependent on the sensors and actuators being used. We propose a reactive layer divided in three modules: the vehicle interface module, the perception module, and the guidance and control module (see Fig. 5). The vehicle interface module contains components named drivers, which interact with the hardware. These drivers are divided in sensor drivers, used to read data from sensors, and actuator drivers, used to send commands to the actuators. Another function provided by the drivers is to convert all the data to the same units and to reference all the gathered data to the vehicle's reference frame. The perception module receives the data gathered by the vehicle interface module (see Fig. 5). Components in the perception module are called processing units. The navigator, the obstacle detector, and the target detectors are its basic components. The navigator processing unit estimates the vehicle position and velocity merging the data obtained from the navigation sensors by means of a Kalman filter [32]. The obstacle detector measures the distance from the robot to the obstacles. Multiple target detectors have been programmed to identify different objects [10]. The guidance and control module includes the vehicle primitives, the coordinator, and the velocity controller. Primitives receive data from the vehicle interface and the perception modules, keeping them independent from the physical sensors and actuators used. The coordinator combines all the responses generated by the primitives in a single one [33]. The velocity controller takes the response provided by the coordinator, turns it into velocity, and generates a force vector for each thruster driver. All the processing units and the primitives can be enabled or disabled by the control execution layer.

Simple primitives can be programmed as simple controllers, however, when the number of parameters increases, it may be difficult to tune them. The proposed control architecture applies RL techniques in the reactive layer. The features of RL make this learning theory useful for robotics since it offers the possibility of learning the primitives in real time.

A. Overview of RL

RL is a very suitable technique to learn in unknown environments. RL learns from interaction with the environment and according to a scalar reward value. The reward function evaluates the environment state and the last taken action with respect to achieving a particular goal. The mission of an RL algorithm is to find an optimal state/action mapping which maximizes the sum of future rewards whatever the initial state is. The learning of this optimal mapping or policy is also known as the reinforcement learning problem (RLP).

The dominant approach over the last decade has been to apply RL using the value function (VF) approach. As a result, many RL-based control systems have been applied to robotics. In [34], an instance-based learning algorithm was applied to a real robot in a corridor-following task. For the same task, in [35], a hierarchical memory-based RL was proposed, obtaining good results as well. In [36], an underwater robot learns different behaviors using a modified VF algorithm. VF methodologies have worked well in many applications, achieving great success with discrete lookup table parameterization but giving few convergence guarantees when dealing with high-dimensional domains due to the lack of generalization among continuous variables [9].

Most of the methods proposed in the RL community are not applicable to high-dimensional systems as these methods do not scale beyond systems with more than three or four degrees of freedom (DoFs) and/or cannot deal with parameterized policies [37]. A particular RL technique, known as the policy gradient (PG) method, is a notable exception to this statement. Rather than approximating a VF, PG techniques approximate a policy using an independent function approximator with its own parameters, trying to maximize the expected future reward [38]. The advantages of PG methods over VF-based methods are numerous. Among the most important are: they have good generalization capabilities which allow them to deal with big state spaces; their policy representations can be chosen so that they are meaningful to the task and can incorporate previous domain knowledge; and often fewer parameters are needed in the learning process than in VF-based approaches [39]. Furthermore, learning systems should be designed to explicitly account for the resulting violations of the Markov property. Studies have shown that stochastic policy-only methods can obtain better results when working in partially observable finite Markov decision processes (POFMDPs) than those obtained with deterministic VF methods [40]. In [41], a comparison between a PG algorithm [42] and a VF method [43] is presented where the VF oscillates between the optimal policy and a suboptimal policy while the PG method converges to the optimal policy.

Starting with the work in the early 1990s [44], [45], PG methods have been applied to a variety of robot learning problems ranging from simple control tasks [46] to complex learning tasks involving many DoFs [47]. PG methods can be used model free, and therefore, they can also be applied to robot problems without an in-depth understanding of the problem or mechanics of the robot [38]. Studies have shown that approximating a policy directly can be easier than working with VFs [39], [41] and better results can be obtained. Informally, it is intuitively simpler to determine *how to act* instead of the *value of acting* [48]. So, rather than approximating a VF, new methodologies approximate a stochastic policy using an independent function approximator with its own parameters, trying to maximize the expected future reward. The purpose is to demonstrate the feasibility of learning algorithms to help AUVs perform autonomous tasks. In particular, this work concentrates on one of the fastest maturing, and probably most immediately significant, commercial application: cable and pipeline tracking.

B. RL-Based Primitives

The features of RL make this learning theory useful for robotics. There are parts of a robot control system which cannot be implemented without experiments. When implementing a reactive robot primitive, the main strategies can be designed without any real test. However, for the final tuning of the primitive, there will always be parameters which have to be set with real experiments. A dynamics model of the robot and environment could avoid this phase, but it is usually difficult to achieve this model with reliability. RL offers the possibility of learning the primitive in real time and avoids tuning it experimentally. RL automatically interacts with the environment and finds the best mapping for the proposed task. The only necessary information which has to be set is the reinforcement function which gives the rewards according to the current state and the past action. It can be said that by using RL the robot designer reduces the effort required to implement the whole primitive, to the effort of designing the reinforcement function. This is a great improvement since the reinforcement function is much simpler and does not contain any dynamics. Another advantage is that an RL algorithm can be continuously learning and, therefore, the state/action mapping will always correspond to the current environment. This is an important feature in changing environments.

RL theory is usually based on finite Markov decision processes (FMDPs) [9]. However, in a robotic system, it is common to measure signals with noise or delays. If these signals are related to the state of the environment, the learning process will be damaged. In these cases, it would be better to consider the environment as a POFMDP. The dynamics of the environment is formulated as a POFMDP and the RL algorithms use the properties of these systems to find a solution to the RLP. Temporal difference (TD) techniques are able to solve the RLP incrementally and without knowing the transition probabilities between the states of the FMDP. In a robotics context, this means that the dynamics existing between the robot and the environment do not have to be known. As far as incremental learning is concerned, TD techniques are able to learn each time a new state is achieved. This property allows the learning to be performed online, which in a real system context, like a robot, can be translated to a real-time execution of the learning process. The term *online* is here understood as the property of learning with the data that are currently extracted from the environment and not with historical data.

The combination of RL with a primitive-based system has already been used in many approaches. In some cases, the RL algorithm was used to adapt the coordination system [49]–[52]. Moreover, some researches have used RL to learn the internal structure of the primitives [53]–[56] by mapping the perceived states to control actions. The work presented by Mahadevan [57] demonstrates that breaking down a robot control policy in a set of primitives simplifies and increases the learning speed. In this paper, policy gradient techniques are designed to learn the internal mapping of a reactive primitive.

C. The Natural Actor–Critic Algorithm

Previous experimental results obtained with basic, PG-based, RL methods in real tasks showed poor performance [58], [59].

Although basic PG methods were able to converge to optimal or near-optimal policies, results suggest that the application of these algorithms in a real task is quite slow and rigid. Fast convergence and adaptation to an unknown, changing environment is a must when dealing with real applications. The learning algorithm selected to carry out the experiments presented in this paper is the natural actor–critic (NAC) algorithm [47]. The NAC algorithm is an extended PG algorithm classified as an actor–critic (AC) algorithm. AC methodologies try to combine the advantages of PG algorithms with VF methods. The algorithm is divided into two main blocks: one concerning the critic and another about the actor. The critic is represented by a VF $V^{\pi}(s)$, which is approximated by a linear function parameterization represented as a combination of the parameter vector v and a particular designed basis function $\phi(s)$, where the whole expression is $V^{\pi}(s) = \phi(s)v$. On the other hand, the actor's policy is specified by a normal distribution $\pi(a|s) = \mathcal{N}(a|Ks,\sigma^2)$. The mean μ of the actor's policy distribution is defined as a parameterized linear function of the form $a = Ks$. The variance of the distribution is described as σ. The whole set of the actor's parameters is represented by the vector $\theta = K$. The actor's policy derivative has the form $\nabla_\theta \log \pi(a|s)$ and to obtain the gradient this function is inserted into a parameterized compatible function approximation $f(s,a)_w^{\pi} = \nabla_\theta \log \pi(a|s)w$, with the parameter vector w as the true gradient. Stochastic PGs allow actor updates while the critic computes simultaneously the gradient and the VF parameters by linear regression.

The parameter update procedure starts on the critic's side. At any time step t, the features $\phi(s)$ of the designed basis function are updated according to the gradients of the actor's policy parameters as shown in

$$\begin{aligned}\tilde{\phi}(s_t) &= [\phi(s_{t+1}), 0]\\ \hat{\phi}(s_t) &= [\phi(s_t), \nabla_\theta \log \pi(a_t|s_t)]\,. \end{aligned} \tag{1}$$

These features are then used to update the critic's parameters and find the gradient. This algorithm uses a variation of the least squares temporal difference (LSTD) (λ) [60] technique called LSTD-Q(λ). Thus, instead of performing a gradient descent, the algorithm computes estimates of matrix A and b and then solves the equation $b + A\tau = 0$ where the parameter vector τ encloses both the critic parameter vector v and the gradient vector w

$$\begin{aligned} z(s_{t+1}) &= \lambda z(s_t) + \hat{\phi}(s_t)\\ A(s_{t+1}) &= A(s_t) + z(s_{t+1})\left(\hat{\phi}(s_t) - \gamma\hat{\phi}(s_t)\right)\\ b(s_{t+1}) &= b(s_t) + z(s_{t+1})r_t\\ [v_{t+1}, w_{t+1}] &= A_{t+1}^{-1} b_{t+1}. \end{aligned} \tag{2}$$

Here, λ is the decay factor of the critic's eligibility $z(s_t)$ [61]; r_t is the immediate reward perceived; and γ represents the discount factor of the averaged reward. On the actor's side, the current policy is updated when the angle between two consecutive gradients is smaller than a given threshold $\epsilon\angle(w_{t+1}, w_t) \le \epsilon$ according to

$$\theta_{t+1} = \theta_t + \alpha w_{t+1}. \tag{3}$$

Here, α is the learning rate of the algorithm. Before starting a new loop, a forgetting factor β is applied to the critic's statistics as shown in the equations below. The value of β is used to increase or decrease the robot's reliance on past actions

$$\begin{aligned} z(s_{t+1}) &= \beta z(s_{t+1}),\\ A(s_{t+1}) &= \beta A(s_{t+1}),\\ b(s_{t+1}) &= \beta b(s_{t+1}). \end{aligned} \tag{4}$$

The algorithm's procedure is summarized in Algorithm 1.

Algorithm 1: NAC algorithm with LSTD-Q(λ)

```
Initialize $\pi(a|s)$ with initial parameters $\theta = \theta_0$, its derivative
$\nabla_\theta \log \pi(a|s)$, and basis function $\phi(s)$ for the value function
$V^{\pi}(s)$. Draw initial state $s_0$ and initialize $z = A = b = 0$.

for $t = 0$ to $n$ do:
    Generate control action $a_t$ according to current policy
    $\pi_t$. Observe new state $s_{t+1}$ and the reward obtained $r_t$.
    Critic Evaluation [LSTD-Q($\lambda$)]
        Update basis functions
        $\tilde{\phi}_t = [\phi(x_{t+1})^T, 0^T]$
        $\hat{\phi}_t = [\phi(x_t)^T, \nabla_\theta \log \pi(u_t|x_t)^T]^T$
        Update sufficient statistics:
        $z_{t+1} = \lambda z_t + \hat{\phi}_t$
        $A_{t+1} = A_t + z_{t+1}(\hat{\phi}_t - \gamma\tilde{\phi}_t)^T$
        $b_{t+1} = b_t + z_{t+1} r_t$
        Update critic parameters:
        $[v_{t+1}^T, w_{t+1}^T] = A_{t+1}^{-1} b_{t+1}$
    Actor Update
        If $\angle(w_{t+1}, w_{t-\tau}) \le \epsilon$, then update policy
        parameters:
        $\theta_{t+1} = \theta_t + \alpha w_{t+1}$
        Forget sufficient statistics:
        $z_{t+1} = \beta z_{t+1}$
        $A_{t+1} = \beta A_{t+1}$
        $b_{t+1} = \beta b_{t+1}$
end
```

At every iteration, action a_t is drawn from current policy π_t generating a new state s_{t+1} and a reward r_t. After updating the basis function and the critic's statistics by means of (LSTD)-Q(λ), the VF parameters v and the gradient w are obtained. The actor's policy parameters are updated only if the angle between two consecutive gradients is small enough compared to an ϵ term. The learning rate of the update is controlled by the α parameter. Next, the critic has to forget part of its accumulated statistics using a forgetting factor $\beta \in [0,1]$. Current policy is directly modified by the new parameters becoming a new policy to be followed in the next iteration, getting closer to a final policy that represents a correct solution of the problem.

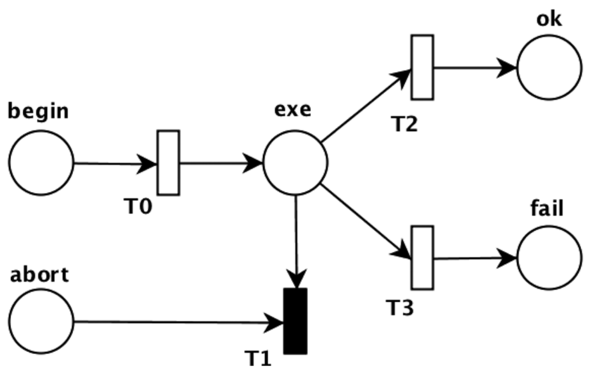


Fig. 6. Example of a task used for the supervision of a primitive.

IV. The Execution Layer

The execution layer acts as an the interface between the reactive and mission layers. This layer translates high-level plans into low-level primitives' execution. Additionally, the execution layer monitors the execution of these primitives and handles the events raised for them. Different approaches are presented in the literature to implement this layer. *Lisp* [62] and *Prolog* [63] interpreters, for example, have been used to translate high-level plans, however, more popular alternatives use state machines [31] or PNs to describe a mission relating the primitives to be executed in each state with the events that produce the transition between those states. The PN formalism has been chosen in this proposal.

One of the first works using PNs in the execution layer was the coordination model for mobile robots [64]. There, a set of PN constructions called Petri net transducers (PNTs) were used to translate the commands generated for the organization level (the mission layer) into something understandable at the execution level (the reactive layer). In marine robotics, the researchers at the Instituto Superior Tecnico (IST, Lisbon, Portugal) developed a control architecture called Coral to be used in the *MARIUS* AUV [65]. The system was based on PNs which were in charge of activating the vehicle primitives needed to carry out a mission. The French AUV *Redermor* [62], designed for military applications of inspection and mine recovery, also uses PNs for modeling the primitives and a Lisp interpreter is employed to execute them in real time. At the Institute of Intelligent Systems for Automation (ISSIA), National Research Council (CNR-ISSIA, Bari, Italy), there is another system based on PNs designed to control an underwater robot [66]. This system uses the PNs not only to describe the execution flow but also to model sensors and controllers. In other fields like the RoboCup, PNs have been also used by several teams [67], [68] for coordination and control purposes.

PNs were invented in 1962 by Carl Adam Petri in his doctoral dissertation [69]. They are one of several mathematical representations of discrete distributed systems. While finite automata can only represent regular languages, PNs are able to describe regular and also nonregular languages. Their algebraic representation is simple and thus makes their specification and analysis simpler. There are several analysis techniques based on their structural and behavioral properties to detect and prevent anomalies and errors [8]. We consider PN structures of the form $N = (P, T, A, W)$ where P, T, and A are a finite set of places, transitions, and directed arcs, and W is a weight function. Arcs link places to transitions and transitions to places but never places or transitions between themselves. Every place can accommodate zero, one, or more tokens. A transition $t \in T$ is said to be enabled if each input place $p \in P$ of t (indicated as $\bullet t$) is marked with at least $w(p,t)$ tokens, where $w(p,t)$ is the weight of the arc from p to t. An enabled transition may or may not fire. A firing of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t,p)$ tokens to each output place p of t (indicated as $t\bullet$), where $w(t,p)$ is the weight of the arc from t to p.

The execution layer presented in this paper is composed of two main components: the architecture abstraction component (AAC) and the Petri net player (PNP) (see Fig. 4). The AAC is located at the bottom of the execution layer and keeps the mission and execution layers, located above, vehicle independent. Hence, the reactive layer below it is the only one bonded to the vehicle's hardware. The AAC offers an interface based on three types of signals: actions, events, and perceptions. Actions are messages transmitted between the AAC and the primitives available in the reactive layer to enable or disable these primitives (e.g., an action can enable a primitive which controls the vehicle's depth pointing to a desired set point and the maximum time to reach it). Events are messages triggered by the primitives in the reactive layer to notify changes in their state (e.g., an event can announce that the desired depth has been reached within the required time or that a failure or a timeout has occurred otherwise). Finally, perceptions are messages containing specific sensor or processing unit values transmitted from the reactive layer to the mission layer to model the world.

The second component, the PNP, executes mission plans defined using the PN formalism by sending and receiving actions and events through the AAC. Then, the PNP controls all the timers associated to timed transitions, fires enabled transitions, sends actions from the execution layer to the reactive layer, and maps the received events in the PN mission plan.

A. Petri Net Building Blocks

PNs have been chosen as a description formalism for several reasons. First, the adoption of this formalism allows us to construct a reliable mission plan, joining small PNs previously evaluated called Petri net building blocks (PNBBs). Following this methodology, it is possible to ensure that the whole mission plan accomplishes a set of required properties. Also, a complex control flow among primitives can be defined using PNBBs. However, for all this to happen, these PNBBs must meet a set of properties. First, all the PNBBs called in a mission must share the same input and output set of places called an interface. A PN interface is composed of a set of input places P_i where $\forall_{p \in P_i} \bullet p = \varnothing$ and a set of output places P_o where $\forall_{p \in P_o} p\bullet = \varnothing$.[1] $P_i \cup P_o$ is used as a fusion place to build more complex structures using other PNBBs. A state that contains a marked place $p \in P_i$ is called an input state while a state that contains a marked place $p \in P_o$ is called an output state. Second property checks reachability. If from all possible input states the PNBB evolves free of deadlocks until reaching an output

[1]$\bullet p$ is the set of transitions with output arcs to place p and $p\bullet$ is the set of transitions receiving input arcs from place p.

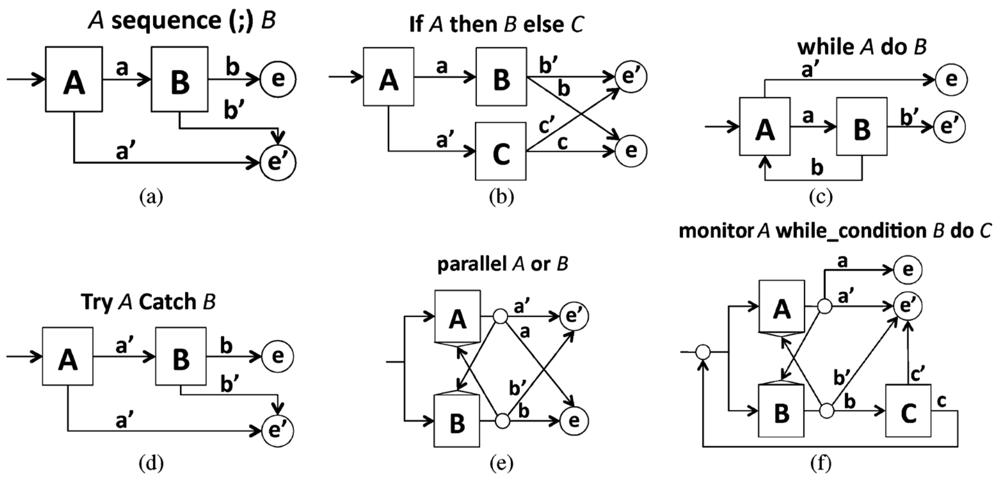


Fig. 7. Schema of the main control structure PNBBs used to control the execution flow between tasks.

state, then the PNBB accomplishes the reachability condition. In the context of this paper, by deadlock we mean the inability to proceed from one state to another due to two primitives, both requiring a response from the other before completing an operation. This situation may happen when two primitives share a resource that can only be used by one of them simultaneously. Finally, to be able to reuse the same PNBB during the execution of a mission, the state of a PNBB, regardless of the places that compose the interface, has to be the same for all the input and output states. These properties can be evaluated through a reachability analysis. Reachability analysis of ordinary PNs can be computationally costly because it depends on the number of places and transitions in the PNs. However, as PNBBs have been designed small enough, no computational burden is expected. Fig. 6 illustrates an example of a PNBB with the interface $P_i =$ *{begin, abort}* and $P_o =$ *{ok, fail}*. There are two different types of PNBBs: tasks and control structures. In the following, we describe their characteristics and usefulness.

1) Tasks: A task is a PNBB which supervises a robot primitive. Tasks interact with the robot reactive layer by means of actions and events through the AAC. Actions are send to the AAC when a particular transition fires while the events, generated by primitives, provoke the firing of enabled transitions. Fig. 6 shows a task able to enable/disable a primitive. The primitive execution is denoted by place *exe*. When this place has a token the primitive is under execution, while when the token is removed the primitive is disabled. Then, the primitive starts when a hierarchically superior PNBB marks the *begin* place and it stops either because the primitive finalizes, sending an event that fires transition $T1$ or $T2$, or when a hierarchically superior PNBB marks the *abort* input place.

2) Control Structures: The PNBBs used to aggregate other PNBBs with the objective of modeling more complex tasks by controlling the flow among different execution paths are called control structures. The resulting net after composing several PNBBs within a control structure is a new PNBB which satisfies the PNBB reachability properties presented before. It is worth noting that when several PNBBs have been composed, it is not necessary to span the whole reachability tree of the resulting PN to ensure the reachability properties. Doing so would have a high computational cost as the complexity of the resultant PN can be very high. These properties are guaranteed by construction and hence a mission plan implemented according to these rules progresses from its starting state to an exit state without coming to a deadlock [70].

Depending on which interface is used, different control structures may be defined. Based on the previously presented interface $I =$ *{begin, abort, ok, fail}*, the following control structures can be implemented.

- *Sequence*: It is used to execute one PNBB after another [see Fig. 7(a)]. If any PNBB finishes with a *fail* (a' or b') the whole structure ends with a *fail* (e'); otherwise, it finalizes with an *ok* (e).
- *If–Then–(Else)*: It executes the PNBB inside the *If* statement [PNBB A in Fig. 7(b)] and depending on whether the PNBB ends with an *ok* or a *fail* the PNBB inside the *Then* statement (PNBB B) or the *Else* statement (PNBB C) if available is executed, respectively.
- *While–Do*: It executes the PNBB inside the *While* statement [PNBB A in Fig. 7(c)]. If this PNBB finishes with an *ok*, it executes the *do* statement (PNBB B); otherwise, it ends with an *ok* (e). If the *do* statement finishes with an *ok*, it executes again the *while* statement; otherwise, it ends the whole structure with a *fail* (e').
- *Try–Catch*: It executes the *Try* PNBB and if it finished with an *ok*. The execution continues after the *Try–Catch* control structure. Otherwise, if it finishes with a *fail*, the *Catch* PNBB is executed, as shown in Fig. 7(d).
- *Parallel–Or*: It executes two PNBBs in parallel. The first structure to finish aborts the other [see Fig. 7(e)]. The *Parallel–Or* finishes with the final state of the first PNBB to end.
- *Monitor–WhileCondition–Do*: It executes the PNBB *Monitor* and *Condition* [PNBBs A and B in Fig. 7(f)] in parallel. If the former finalizes, the latter is aborted and the output is the former's output. Otherwise, if the latter finalizes first, the *Monitor* PNBB is aborted and the *Do* statement (PNBB C) is executed. When the *Do* block finalizes, blocks *Monitor* and *Condition* are executed again.

The composition of a mission by adding tasks and control structures is exemplified using Figs. 6, 8, and 9. Two tasks, *Task1* and *Task2*, each one described by a PNBB like the one

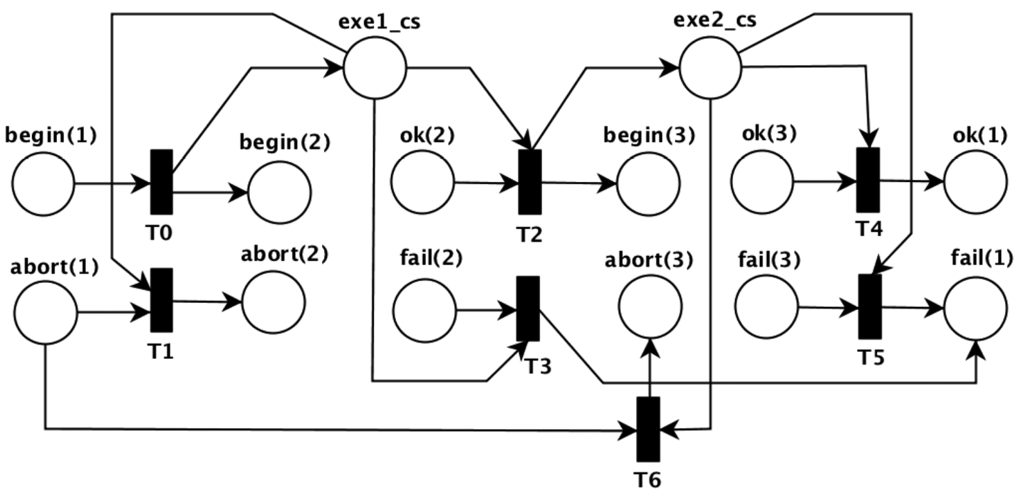


Fig. 8. Control structure used to sequence two PNBBs.

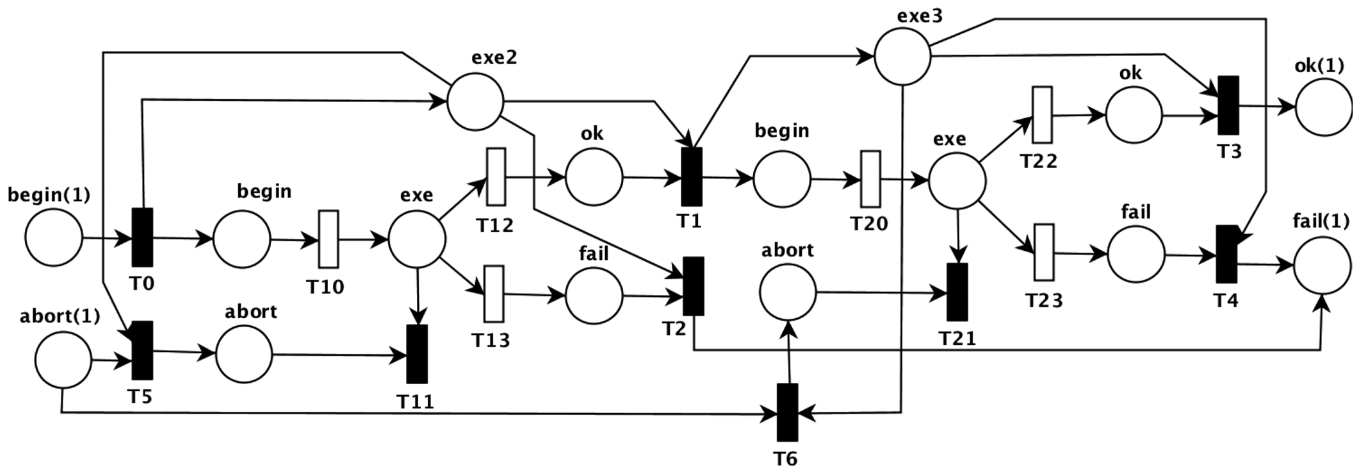


Fig. 9. Composition of two PNBBs with a sequence control structure. The resulting PNNB can also be composed of other control structures.

presented in Fig. 6, are attached to a sequence control structure as the one shown in Fig. 8. The resulting PN is depicted in Fig. 9. In this process, the interface *{begin, abort, ok, fail}* of *Task1* and *Task2* has been composed of interfaces 2 and 3 of the sequence control structure, respectively. Interface 1 of the resulting PNBB can also be used in bonding this sequence of tasks with another hierarchically superior control structure.

V. The Mission Layer

Predefined plans allow scientists to collect data where and when they want. However, when dealing with unknown or changing environments with imprecise and noisy sensors, predefined offline plans can fail. The difficulty of controlling the time in which events happen, energy management, sensor malfunctions, or the lack of onboard situational awareness may cause offline plans to fail during execution as assumptions upon which they were based are violated [6]. Therefore, it is worth the effort to study the inclusion of an onboard planner with the ability to modify or replan the original plan when dealing with missions in which an offline plan is susceptible to failure. Thus, a compromise between predefined offline plans and automatically generated online plans is desirable. Our solution starts with the introduction of a high-level language, named Mission Control Language (MCL), to describe offline plans that can be automatically compiled into a PN following the properties previously introduced. Next, the inclusion of an onboard planner to add deliberative capabilities to the system has been studied as well.

A. The MCL

Several languages have been introduced in the literature to describe the sequence of tasks to undertake when carrying out a mission. Esterel [75] is a well-known synchronous language for reactive programming. Its formal approach allows it to compile programs written in Esterel into efficient FSMs that can be verified *a posteriori*. The presented proposal presents some benefits with respect to Esterel: the PN formalism allows a more compact representation of a DES and it is better suited to express parallelism than the FSM formalism. Moreover, instead of verifying the resulting FSMs, MCL builds a mission plan by composing some blocks that accomplish a set of properties ensuring that the resultant mission is verified by construction. Finally, noninstantaneous deadlocks are not statically detected in Esterel while they can be avoided with our proposal. On the other hand, Esterel is a more general purpose language than the one presented in this paper, which is rather limited to mission descriptions. Another well-known language is the Task

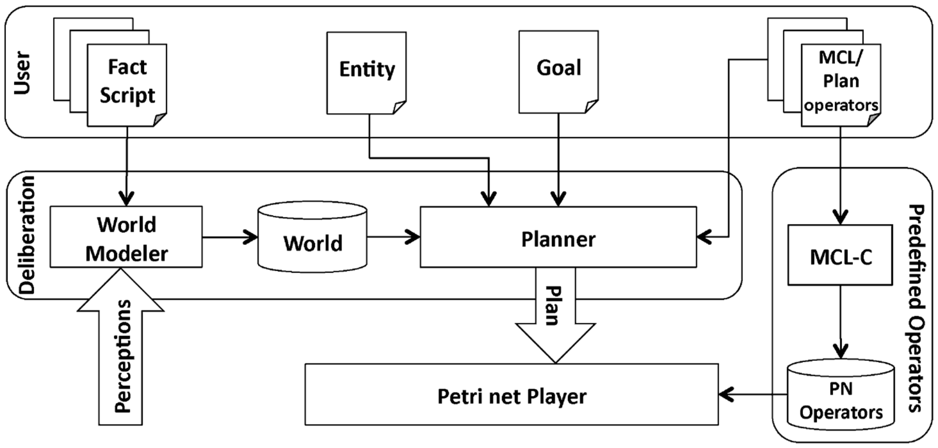


Fig. 10. Deliberation components used to build online plans.

Description Language (TDL) [76]. TDL allows defining a mission and compiling it into a pure C++ file executed on the vehicle by means of a platform-independent task management library. TDL has been designed to simplify the function control at the task level. For this purpose, TDL uses multithreading code which is often difficult to understand, debug, and maintain using C++. In our proposal, if the description of a mission requires dealing with parallel execution, parallel PNBBs can be used to easily overcome this problem. The TDL is based on task trees that can be dynamically modified. Each node corresponds to a basic action or a goal that is expanded using other task trees. Also, the Reactive Action Package (RAP) [77] is another popular language. In RAP, each package contains a goal and a set of task nets to achieve this goal. Each task net contains basic actions or goals that must be achieved using other packages. Planning techniques are needed to know which task net is more suitable to achieve a particular package goal.

While some languages introduce verification capabilities, others introduce planning skills. This proposal deals with these two issues separately. On the one hand, the MCL allows the construction of a PN under the assumption of a formal set of requirements. On the other hand, a planner can be added to decide which piece of predefined MCL code (i.e., which sub-PN mission) has to be executed to achieve a specific goal.

Since the direct manipulation and construction of a PN mission becomes rapidly cumbersome for complex missions, a high-level language able to automatically compile into a PN has been implemented. First, the actions and events that can be communicated through the AAC toward the reactive layer must be described. Then, to define the PNBBs, it is necessary to specify their internal structure: places, transitions, and arcs. Finally, once the tasks and the control structures have been described, a mission plan can be encoded. This is the only part that must be rewritten for every new mission. When encoding a mission in MCL, control structure PNBBs can be seen as standard flow control instructions used in other languages but with a higher degree of parallelism, while tasks act as function calls.

The process of generating a PN mission from an MCL program is automatically performed by the MCL compiler. The generated code contains a single PN describing the mission plan. It is encoded using a standard Extensible Markup Language (XML)-based format for a PN called Petri Net Markup Language (PNML) [78]. The complete definition of the MCL language and how the MCL compiler translates it into a PN can be found in [79].

B. Onboard Planning

Not many successful approaches using deliberative modules onboard an AUV are found in the literature. Researchers at the Ocean Systems Laboratory, Heriot-Watt University (Edinburgh, U.K.) have developed an architecture with planning abilities to carry out multiple AUV missions [71]. They used online plan repairing algorithms [72] to obtain automated plans similar to the ones defined by the user. The T-REX [73], developed by the Monterey Bay Aquarium Research Institute (Moss Landing, CA), is another AUV architecture that takes advantage of an onboard planner. It is a part of the remote agent architecture [74] developed by the National Aeronautics and Space Administration (NASA) Deep Space 1 spacecraft. Another approach in this area is the Orca project [6] under which an intelligent mission controller for AUVs was developed that provides a context-sensitive reasoner that enables autonomous agents to respond quickly, automatically, and appropriately to their context.

To add onboard planning capabilities to COLA2, two components are included in the mission layer: a world modeler and a planner (see Fig. 10). The first receives perceptions through the AAC and, applying a set of scripts defined by the user, models the world. The second is a standard domain-independent planner algorithm which, given a world model and a list of available planning operators, generates a plan that fulfill the goals described by the user.

In classical planning, a planning operator is a basic action that can be autonomously executed by an AUV. Each planning operator is defined as a triple $o =$ *{name, preconditions, effects}* where the *name* is defined as *name* $= n(x_1, \ldots, x_k)$, n is a symbol called the operator name, and $x_1, \ldots, x_k$ are all the variables that appear anywhere in o. We call these variables entities and each entity is assigned a type. Facts are used to describe the other two elements of an operator. They are the combination of one or two entities with a proposition representing a quality or a relation between the entities. *Preconditions* indicate the facts that must be present in the world model to apply the operator. Finally, *effects* are the changes that should be produced in the

TABLE I
PLANNING OPERATOR

Goto planning operator
name: GotoOp *Robot* r, *Zone* o, *Zone* d
pre: r in o, r position_ok, r no_alarms
del: r in o
add: r in d
exp: o != d
mcl: GotoMCL(*d*)

world model when the operator is applied. Effects are separated between *add* and *del* effects: *add* effects are the facts that should be added into the world model after applying the operator and *del* effects are the facts that should be removed from it once the operator is applied. Additionally, a set of boolean expressions can be added to the operator. The operator can be executed only when all the expressions are true. If no cost value is specified, the planner tries to obtain the plan with the shortest sequence of planning operators possible.

Instead of describing one planning operator for each vehicle primitive, each planning operator describes a piece of MCL code used for solving a particular phase of a more complex mission. This solution makes the robot's behavior more predictable than an arbitrary combination of primitives selected by the onboard planner. Moreover, although the proposed planner only sequences the planning operators to be executed, parallelization, or iterative execution of primitives, for example, can be performed inside the MCL. A positive cost value associated to each planning operator is used by the planner to find the lowest cost combination of operators which accomplish all the goals. Time and resources are not taken into account when planning, for simplicity reasons. These simplifications increase planning speed and avoid having to predict the state of the resources and, as usually happens when planning with resources, taking decisions based on the worst possible case. As a drawback, the mission plan fails more often, and the AUV has to replan the whole mission when the facts generated by the world modeler do not match with those expected by the generated plan. However, it is preferable to generate new plans faster than generate slower but more reliable plans because even these ones fail when dealing with a dynamic environment. Additionally, this approach simplifies the utilization of offline planning algorithms. Therefore, each planning operator has to include: preconditions, effects, a set of boolean expressions, and a link to the MCL piece of code to be executed. Table I shows an example of a planning operator that describes the action of moving an AUV from an initial zone to another.

The planner used in the current implementation uses a simple breadth-first search algorithm which works in the state space using the restricted planning conceptual model [80]. The planner's output is a plan containing the ordered sequence of planning operators to be executed and the sequence of facts that should be present in the world model while executing that plan. If, when the plan is under execution, the facts generated by the world modeler component do not match with the facts expected in the plan, the plan is aborted and a new one is computed.

TABLE II
FACT PROVIDER SCRIPT

low_battery script
id: low_battery
pre: auv.battery < 30
del: Robot::auv BatteryOk
add: Robot::auv BatteryLow

The second component required to plan is the world modeler. Offline planners typically use a database of known facts about the world to plan their actions. However, when dealing with a dynamic environment, several techniques have been studied to get information from the real world and translate it into a set of facts. A world modeler, inspired by the *context provider* presented in [81], has been implemented in this proposal. To extract a collection of facts from the world model, the user defines a set of scripts whose input parameters are the perceptions received from the reactive layer.

Table II shows one of these scripts. Each script is defined by the triple $s = \{name, preconditions, effects\}$. The *name* is an identifier. The *preconditions* are a set of boolean expressions relating perceptions, coming from the reactive layer, and literal values defined by the user. *Effects* are the changes to be produced in the world model if the script is applied. They are also divided between *add* and *del* effects. When all the conditions inside one script are true, the effects are applied. When the *auv.battery* perception is sent through the AAC to the world modeler component, the script *low_battery* is executed. If the value of *auv.battery* is smaller than 30%, the fact *auv_BatteryOk* is removed from the world database while the fact *auv_BatteryLow* is added to it.

Note that the world modeler scripts modify the world model and not the planning operators. Planning operators only indicate which are the more likely changes to be produced in the world model if the operator is applied. This information is used by the planner when building the plan but is not used to modify the world model when the plan is under execution.

VI. Experimental Setup

The purpose of this section is to report the main characteristics of the elements used to build the experimental setup. It describes the application of the proposed architecture to the *Ictineu* AUV [10] in the context of a cable tracking mission scenario. Initially, the most notable features of the vehicle *Ictineu* are given. Then, an insight of the problem of underwater cable tracking together with the vision-based system used is also introduced. Finally, the underwater cable tracking mission is programmed using both possible alternatives provided by the proposed architecture. First, the MCL is used to describe a predefined mission and, second, the onboard planner automatically decides which planning operators execute during the mission.

A. *Ictineu AUV*

The *Ictineu* AUV is a research vehicle built in the Computer Vision and Robotics Group (VICOROB), University of

Girona (Girona, Spain), which constitutes the experimental platform of this work. The experience obtained from the development of previous vehicles in the group, Garbi ROV [82], Uris [83], and Garbi AUV, made it possible to build a low-cost vehicle of reduced weight (52 kg) and dimensions (74 $\times$ 46.5 $\times$ 52.4 cm^3) with remarkable sensorial capabilities and easy maintenance. *Ictineu* AUV was conceived around a typical open frame design [84]. Although a majority of AUVs systems are shaped like a torpedo for speed and coverage purposes, the *Ictineu* AUV was initially designed aiming for high maneuverability in close spaces. Therefore, the *Ictineu* AUV is propelled by six thrusters that allow it to be fully actuated in *Surge* (movement along X-axis), *Sway* (movement along Y-axis), *Heave* (movement along Z-axis), and *Yaw* (rotation around Z-axis) achieving maximum speeds of 3 kn. *Ictineu* AUV is passively stable in both *pitch* and *roll* DoFs as its center of buoyancy is above the center of gravity, separated by a distance of 24 cm, and thus, giving great stability to the vehicle in these DoFs. The onboard embedded computer has been chosen as a tradeoff between processing power, size, and power consumption. An ultralow-voltage (ULV) Core Duo processor with the 3.5-in small form factor was selected. All *Ictineu*'s equipment is powered by two battery packs. The first one, at 12 V, powers the computer, the electronics, and the sensors, while the second one, at 24 V, provides power to the thrusters. Each battery pack has a 10-Ah capacity, which allows for an autonomy of 2.5 h of intense running activity.

For navigation purposes, *Ictineu* is equipped with a Doppler velocity log (DVL) specially designed for applications which measure ocean currents and vehicle speed over ground, and as an altimeter using its three acoustic beams. A compass which outputs the sensor heading (angle with respect to the magnetic north), a pressure sensor for water-column pressure measurements, a low-cost miniature attitude and heading reference system (AHRS), which provides a 3-D orientation (attitude and heading), 3-D rate of turn as well as 3-D acceleration measurements and a Global Position System (GPS) that is only available when the vehicle is on the surface. An extended Kalman filter (EKF) [32] is the sensor fusion algorithm used to combine the data gathered by all these sensors and estimate the *Ictineu*'s position, orientation, and velocity. Finally, the robot is also equipped with two cameras. The first camera is a forward-looking color camera mounted on the front of the vehicle and intended for target detection and tracking, inspection of underwater structures, and to provide visual feedback when operating the vehicle in ROV mode. The second camera is located in the lower part of the vehicle and is downward looking. This camera is mainly used to capture images of the seabed for research on image mosaicing. The downward-looking camera is the main sensor used for the cable tracking primitive detailed next.

B. Cable Tracking in Underwater Environments

The use of professional divers for the inspection and maintenance of underwater cables/pipelines is limited by depth and time. ROVs represent an alternative to human divers. The main drawback of using ROVs for surveillance missions resides in their cost, since it increases rapidly with depth, because of the requirements for bigger umbilicals and support ships. Also, a manual visual control is a very tedious job and tends to fail if the operator loses concentration. Although for a particular missions the use of a ROV could be more profitable (low deep or close coast missions), for the kind of missions like the one presented in this paper the advantages of an AUV are clear. Cable tracking missions require a vehicle with long-range capabilities. An AUV can perform all the tracking missions by itself gathering all useful data from sensors and the surface at the desired location for recovery.

Several systems have been developed for underwater cable/pipeline inspection purposes. Basically, the applied technology classifies the methodologies in three big groups depending on the sensing device used for tracking the cable/pipeline: magnetometers- [85], [86], sonar- [87], [88], and vision-based methods [89]. Compared to magnetometer or sonar technology, vision cameras, apart from being a passive sensor, provide far more information with a larger frequency update, are inexpensive and much less voluminous, and can be powered with a few watts. Light-emitting diode (LED) technology also contributes to reducing the size of the lighting infrastructure and the related power needs, which also matters in this case. The main drawbacks of using a camera are its short-range capabilities compared to the sonar and its high susceptibility to environmental factors. Given the good results obtained by the vision-based algorithm in real underwater conditions [90], a vision camera has been the technology chosen to track the cable. For performing a camera vision-based tracking, the AUV control architecture must command the vehicle so as to let it fly over the cable/pipeline, which leads to the tracking, frame by frame, of its position and orientation, to confine it within the field of view of the camera during the mission. The mission proposed in this paper consists of building a georeferenced photomosaic [91] of an underwater cable. The vehicle must be able to search for the underwater cable, follow it while gathering images, and keep an accurate positioning during the mission. Then, combining the vehicle's navigation data with the acquired images, the mosaic is composed offline. For these experiments, a vision-based system tested and developed at the University of the Balearic islands [90] has been chosen to track a submerged cable in a controlled environment. The vision algorithm processes the image in real time and computes the polar coordinates (ρ, Θ) of the straight line corresponding to the detected cable in the image plane (do not confuse the parameter's vector of the actor's policy, defined in Section III-C, θ with the polar coordinate of the cable in the image plane Θ). When (ρ, Θ) are the parameters of the cable line, the Cartesian coordinates (x, y) of any point along the line must satisfy

$$\rho = x\cos(\Theta) + y\sin(\Theta). \tag{5}$$

As shown in Fig. 11, (5) allows us to obtain the coordinates of the cable intersections with the image boundaries (X_u, Y_u) and (X_L, Y_L), thus the midpoint of the straight line (x_g, y_g) can be easily computed $(x_g, y_g = X_L + X_u/2, Y_u + Y_L/2)$. The computed parameters (ρ, Θ, x_g, y_g) together with their derivatives are sent to the guidance and control module to be used by the

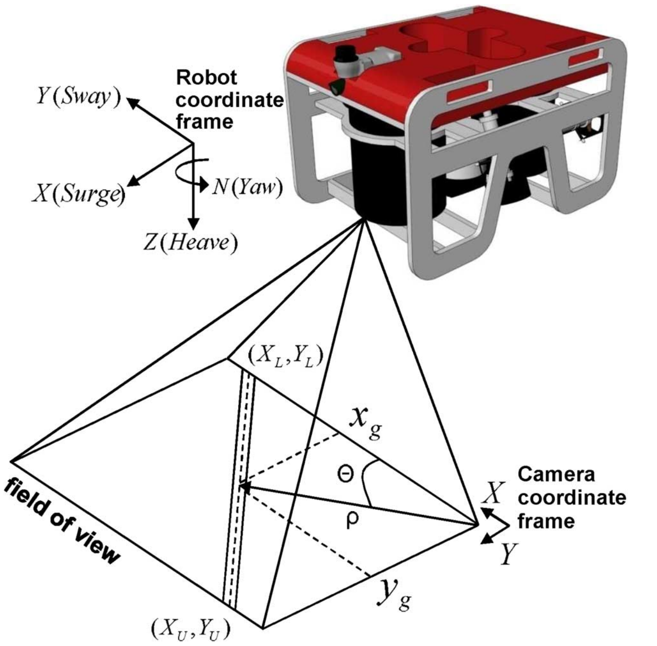


Fig. 11. Coordinates of the target cable with respect to the *Ictineu* AUV. The image shows the calculation of the polar coordinates (ρ, Θ) within the image plane.

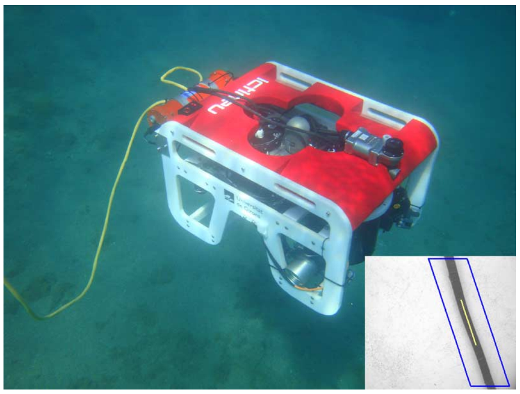


Fig. 12. *Ictineu* AUV in the test pool. An artificial image of a real seafloor is placed at the bottom. Small bottom-right image: Underwater cable detection performed by the vision algorithm.

learning algorithm. Fig. 12 shows a real image of *Ictineu* AUV while detecting a cable.

C. Vehicle Primitives

Several primitives have been programmed to implement the cable tracking mission. Next, the main ones are introduced.

- *cableTracking*: This is the main primitive for the proposed mission. Whenever the cable is within the field of view of the downward-looking camera, using a stimulus-to-action mapping formerly learned in simulation and continuously adapted during execution, this primitive guides the robot to follow the cable. It controls the vehicle in both surge and yaw DoFs, as shown in Fig. 13(a).
- *goto*: It is a simple line-of-sight (LOS) algorithm with cross-tracking error [92]. It is used to guide the robot toward the desired waypoints. It uses the localization data provided by the navigator and controls the path in both the surge and yaw DoFs.

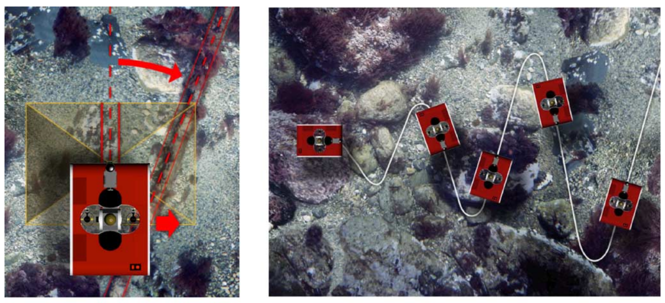


Fig. 13. (a) Cable tracking primitive which controls the AUV sway and yaw DoFs (red arrows). (b) The search pattern primitive which performs an incremental zig–zag movement until a timeout is reached or the task is aborted otherwise.

- *altitude*: This primitive controls the robot in the heave DoF. It reads the altitude data from the obstacle detector component and uses a simple controller to achieve and keep a constant distance with respect to the seafloor.
- *searchPattern*: It performs a search trajectory controlling the surge and yaw DoFs. Given an initial orientation, the incremental search pattern shown in Fig. 13(b) is executed.
- *cableDetector*: It indicates whether the underwater cable is inside the camera field of view. This primitive is used together with the *searchPattern* to find the underwater cable.
- *invalidPositioning*: An EKF [32] is the sensor fusion algorithm used to combine the data gathered by all the navigation sensors included in *Ictineu* AUV. The filter is initialized with the data gathered by the GPS, therefore, all the images captured by the camera sensor can be georeferenced using the navigation data. However, when the vehicle is submerged, no GPS data are available and thus the EKF relies only on incremental measures sensed by the DVL. Because DVL measures are noisy, the uncertainty in the vehicle's position starts growing. The *invalidPositioning* primitive raises an event when this uncertainty is higher than a threshold.
- *getPositionFix*: It waits until the GPS receives some valid data packets. These data are then used to update the navigation EKF reducing the vehicle's position uncertainty.
- *alarmDetector*: It is responsible for the control of temperature and pressure inside the vessels. It also checks for water leakages and battery levels.

D. NAC Algorithm Configuration for the cableTracking Primitive

From all the primitives described in the previous section, the *cableTracking* primitive is the one selected to be learned. This primitive is considered the most complex one as it will be subjected to continuous environmental changes (cable position, thickness, changing illumination conditions, etc.). As described in the algorithm learning procedure of Algorithm 1, the *cableTracking* primitive receives the state of the environment as input. The state is represented by a 4-D vector containing four continuous variables $(\Theta, \delta\Theta/\delta t, x_g, \delta x_g/\delta t)$ where Θ is the angle between the Y-axis of the image plane and the cable, x_g is the X-coordinate of the midpoint of the cable in the

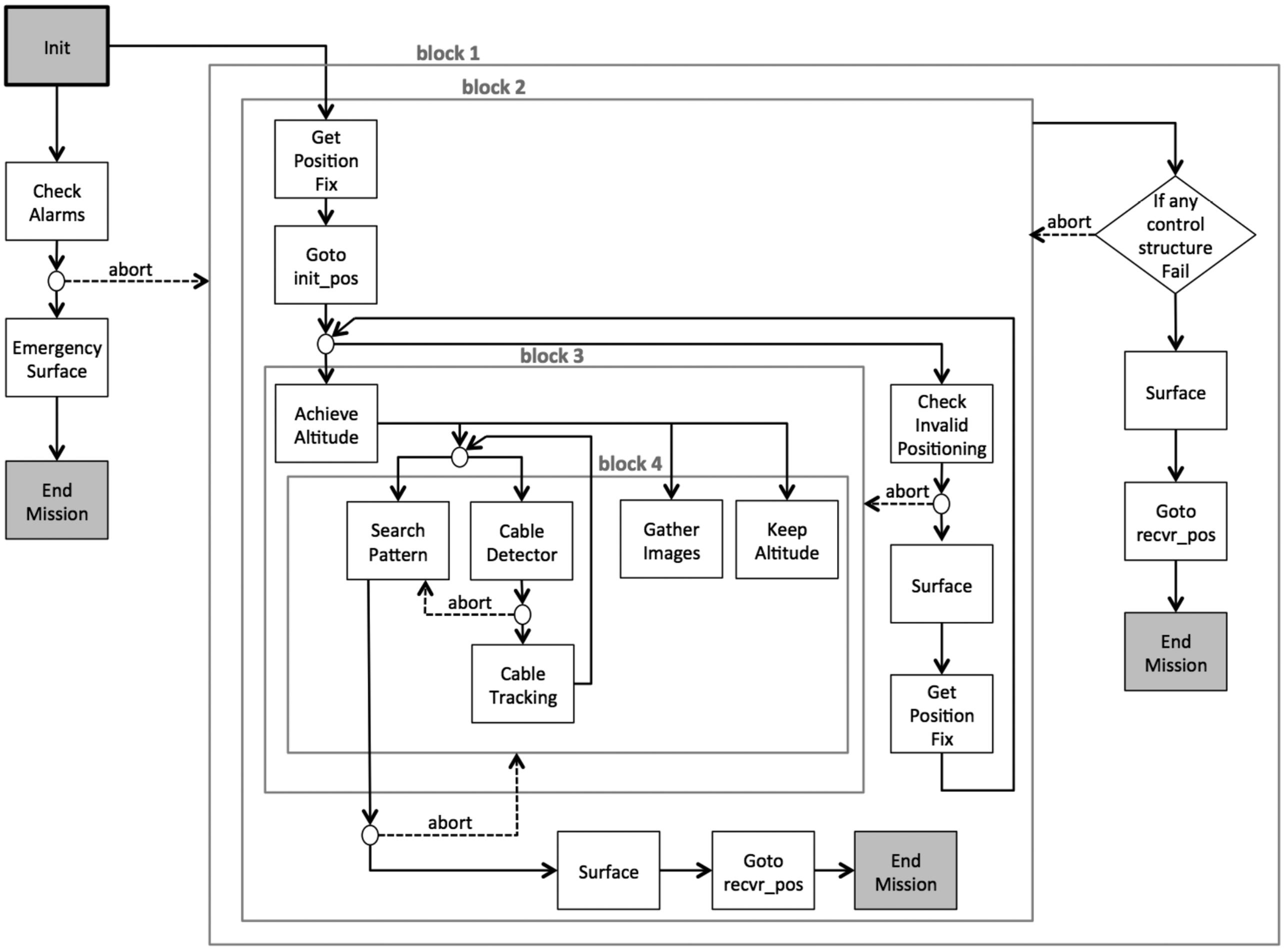


Fig. 14. Predefined cable tracking mission flowchart.

image plane, and finally, $\delta\Theta/\delta t$ and $\delta x_g/\delta t$ are Θ and x_g derivatives, respectively. The bounds of these variables are $(-\pi/2)$ rad $\leq \Theta \leq \pi/$ 2 rad; -1 rad/s $\leq \delta\Theta/\delta t \leq +$ 1 rad/s; $-0.539 \leq x_g \leq 0.539$ (the value of x_g is initially computed in pixels; these bound values are scaled from the size of the image in pixels); and $-0.5 \leq \delta x_g/\delta t \leq +0.5$ (bounds computed from the maximum and minimum values of the derivatives of x_g). From these input states, the *cableTracking* primitive makes decisions concerning two DoFs: Y movement (sway) and Z-axis rotation (yaw). Therefore, the continuous action vector is defined as $(a_{\text{sway}}, a_{\text{yaw}})$ with boundaries $-1 \leq a_{\text{sway}} \leq +1$ and $-1 \leq a_{\text{yaw}} \leq +1$. The X movement or *surge* of the vehicle is not learned. A simple controller has been implemented to control the X DoF. Whenever the cable is centered in the image plane (the cable is located inside the 0 reward boundaries), the robot automatically moves forward ($a_{\text{surge}} = 0.3$ m/s). If the cable moves outside the *good* limits or the robot misses the cable, it stops moving forward ($a_{\text{surge}} = 0$ m/s).

The main policy has been split into two subpolicies. a_{sway} actions cause Y displacements on the robot, and therefore, X displacements on the image plane. Then, it can be easily noticed that a_{sway} actions directly affect the position of x_g along the X-axis of the image plane together with its derivative. In the same way, a_{yaw} causes rotation around Z-axis, and therefore, Θ angle variations in the image plane. Although learning uncoupled policies certainly reduces the overall performance of the robot, this method greatly reduces the total number of stored parameters, making it easier to implement. Actor policies for yaw and sway are described as normal Gaussian distributions of the form $\pi(u|x) = N(u|Kx, \sigma^2)$, where the variance σ^2 of the distribution is fixed at 0.1. Additional basis functions for sway and yaw critic parameterization are chosen as $\phi(x) = [x_1^2, x_1x_2, x_2^2, 1]$, which experimentally have proven to be good enough to represent both functions.

A continuous function has been designed to compute specific rewards for each learned policy. Rewards are given by $r(x_t, u_t) = x_t^T Q x_t + u_t^T R u_t$ with $Q_{\text{sway}} = \text{diag}(1.1, 0.2)$ and $R_{\text{sway}} = 0.01$ for sway DoF and $Q_{\text{yaw}} = \text{diag}(2, 0.25)$ and $R_{\text{yaw}} = 0.01$ for yaw DoF. $Q's$ and $R's$ values have been tuned experimentally. The learning parameters have the same value for both policies. The learning rate has been fixed to $\alpha = 0.01$ and the decay factor of the eligibility has been set to $\lambda = 0.8$, both tuned experimentally. A discount factor for the averaged reward is set to $\gamma = 0.95$ and a forgetting factor for the matrices A and b has been set to $\beta = 0.9$. For every iteration, the policy is updated only when the angle between two consecutive gradient vectors accomplishes $\angle(w_{t+1}, w_t) \leq \epsilon = \pi/180$.

E. The Mission Program

The mission to execute has been programmed using two paradigms: first, an offline mission plan coded in MCL, and second, using an onboard planner able to combine a set of

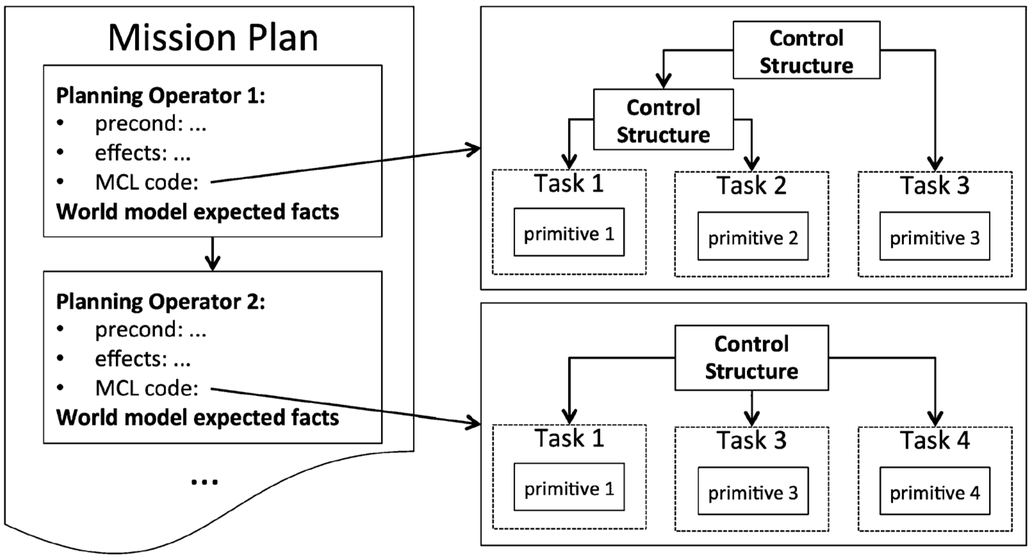


Fig. 15. Example of a mission plan obtained using an onboard planner and its relation with control structures, tasks, and primitives.

TABLE III
PIPE TRACKING MISSION PLANNING OPERATORS

Take Position Fix
op: TakeFixOp *Robot* r
pre: r surface, r position_bad, r no_alarms
add: r position_ok
del: r position_bad
mcl: TakeFixOp()
Search Cable
op: SearchCableOp *Robot* r *Object* o
pre: r seafloor, r position_ok, r no_alarms, r in inspection, o not_found
add: o found
del: o not_found
mcl: SearchCableOp(*altitude*, *timeout*)
Cable Tracking
op: CableTrackingOp *Robot* r *Object* o
pre: r position_ok, r no_alarms, o found, o not_mapped
add: o mapped
del: o not_mapped
mcl: CalbeTrackingOp(*altitude*, *params_NAC*)

planning operators. Fig. 14 shows a flowchart of the MCL mission programmed for the first solution. The mission starts after initializing the vehicle (*Init*), taking a GPS position fix (*GetPositionFix*) and driving the vehicle to the starting position [*Goto(init_pos)*]. Then, the vehicle submerges until an altitude of 1 m with respect to the seafloor is achieved (*AchieveAltitude*). Thereafter, a *SearchPattern* task together with a *CableDetector* task are enabled while gathering images and keeping the current altitude. If the *CableDetector* task successfully finds the cable, it finalizes aborting the *SearchPattern* task and enabling the *CableTracking* task. When the *CableTracking* task misses the cable, the *SearchPattern* and *CableDetector* tasks are enabled again. However, if the *SearchPattern* task finishes because it is unable to find the cable before the timeout, the whole block 4 is aborted and the vehicle surfaces and goes to the recovery position to be recovered. To keep the vehicle always localized, if the *CheckInvalidPositioning* task raises an *ok* event, block 3 is aborted and the *Surface* and *GetPositionFix* tasks are executed. Because block 3 is encoded in MCL using a *monitor-while* control structure, once the vehicle is correctly positioned, block 3 is executed again. The whole mission is inside a *try-catch* control structure, represented by block 2. If any of the tasks within this block finalize unexpectedly (in the *fail* state), the catch block is executed aborting block 2 and surfacing the vehicle and driving it to the recovery zone. Moreover, parallel to all this code, a *CheckAlarm* task is under execution. If any event checked by this task rises, block 1 which is encoded with a *monitor* control structure is aborted and the vehicle surfaces by means of a drop-weight emergency system.

If the mission is programmed by means of an onboard planner, a set of planning operators must be first defined specifying its preconditions, effects, and the MCL code to be executed for each operator. For this mission, seven planning operators have been used: *GotoOp*, *SurfaceOp*, *AchieveAltitudeOp*, *TakeFixOp*, *CheckAlarmOp*, *SearchCableOp*, and *CableTrackingOp*. The most complex operators are described in Table III. We must not confuse the robot's primitives (i.e., *goto*) that are the *processes* running in the reactive layer trying to achieve a goal, with the tasks, which are PN structures that supervise the execution of primitives (i.e., *Goto*), and the planning operators used to plan (i.e., *GotoOp*). Fig. 15 shows the relation between all the elements involved in the execution of a plan. A mission plan is a sequence of planning operators plus a list of the facts expected in the world model for each operator. Each planning operator contains a link to a piece of the MCL code. This code is a single PN, predefined by the user, used to achieve a particular phase of a mission. These pieces of the MCL code contain task PNBBs that supervise the execution of primitives in the reactive layer, and control structure PNBBs, defining the execution flow between the PNBBs.

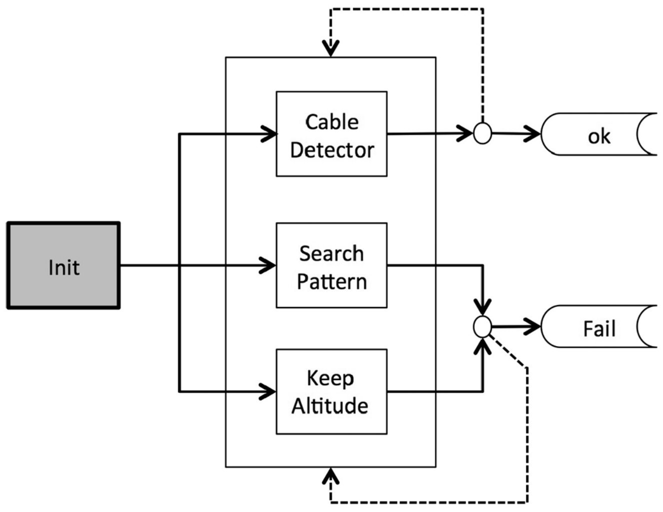


Fig. 16. MCL code flowchart associated to the *SearchCableOp* planning operator.

TABLE IV
FACTS THAT CAN BE GENERATED DURING THE MISSION EXECUTION BY THE WORLD MODELER SCRIPTS

Fact	WM Script	Primitive	W_0/Goal
robot position_bad	navigation_status	InvalidPositioning	W_0
robot position_ok	navigation_status	InvalidPositioning	–
robot surface	robot_altitude	Navigator	W_0/Goal
robot seabottom	robot_altitude	Navigator	–
robot no_alarms	alarms_status	Alarm	W_0
robot alarms	alarms_status	Alarm	–
robot in deploy	robot_position	Navigator	W_0
robot in inspection	robot_position	Navigator	–
robot in recovery	robot_position	Navigator	Goal
cable not_found	cable_status	SearchCable	W_0
cable found	cable_status	SearchCable	–
cable not_mapped	mission_status	SearchCable	W_0
cable mapped	mission_status	SearchCable	Goal

An example of the flowchart of the MCL piece of the code associated to the planning operator *SearchCableOp* is shown in Fig. 16. In this figure, three tasks are executed in parallel. If the *CableDetector* task finds the cable, then this fact will be added to the world model, as expected in the plan, and the *SearchCableOp* operator finishes in the *ok* state continuing the execution of the mission plan previously computed. However, if the planning operator finishes in the *fail* state because of a timeout in the *SearchPattern* or *KeepAltitude* tasks, the *SearchCableOp* task is aborted adding an unexpected fact into the world model and forcing the computation of a new plan.

The world modeler is responsible for keeping the facts which describe the world up to date. Table IV shows a list with some of the facts in the world model, the scripts to generate each one of these facts, the primitives affected, and if these facts can be in the initial world model (W_0) or in the mission goal state.

Once a mission plan is generated, it is composed of a sequence of planning operators with the facts that should be present in the world model before and after the operators' execution. If any operator finishes with a *fail*, or the facts that the world modeler adds in the world model do not match with the ones expected by the planner, a new plan is built from the current situation. The initial plan obtained in the proposed cable tracking mission is composed of a sequence of seven planning operators: *TakeFixOp()*, *GotoOp(inspection)*, *AchieveAltitudeOp(altitude)*, *SearchCableOp(altitude, timeout)*, *CableTrackingOp(altitude, param_NAC)*, *SurfaceOp()*, and *GotoOp(recovery)*. However, several situations can trigger a world modeler script adding a new fact in the world model. Some of these situations are: losing the cable, raising an alarm, or an uncertainty of the vehicle's position beyond a predefined threshold. If this new fact does not coincide with the ones expected in the mission plan, a new plan is generated by the onboard planner and the previous one is aborted. Programming a mission using planning operators and world modeling scripts can be less intuitive than coding an offline imperative mission using MCL. However, for complex missions where a large number of events may take place, the use of an onboard planner simplifies the mission description and avoids possible errors introduced by the user.

VII. RESULTS

The results presented hereafter have been organized into three phases. Initially, the primitive being used in the proposed mission, the *cableTracking* primitive, has been tested in a water pool using the *Ictineu* AUV. With the aim of speeding up the learning process, first, the robot interacts with a computer simulator and builds an initial policy for this RL-based primitive. Once the simulated results are accurate enough and the primitive has acquired enough knowledge from the simulation to build a safe policy, the learned-in-simulation policy is transferred to the *Ictineu* AUV. Then, in a second phase, the primitive is improved in a real environment. More details on the simulation process can be found in [93]. Finally, the third phase illustrates simulated results of the whole system including the execution and mission layers.

A. Phase One: Simulated Learning Results

The computed model of the underwater robot *Ictineu* AUV navigates a 2-D world model at a 0.8-m height above the seafloor. The simulated cable is placed on the bottom in a fixed position. The robot has been trained in an episodic task. An episode ends either after 150 iterations or when the robot misses the cable in the image plane, whatever comes first. When the trial ends, the robot position is reset to a random position and orientation around the cable's location, allowing any location of the cable within the image plane at the beginning of each trial. According to the value of the state and action taken, a scalar immediate reward is given at each iteration step.

The experiments in the simulation phase have been repeated in 100 independent runs. At the beginning of each run, the policy parameters are randomly initialized for each one of the policies and the learning procedure starts. The effectiveness of each episode is measured as the total reward per episode perceived by the current policy. Fig. 17(a) shows the learning evolution of the sway policy applying the NAC algorithm as a mean of 100 independent runs. For comparative purposes, the learning

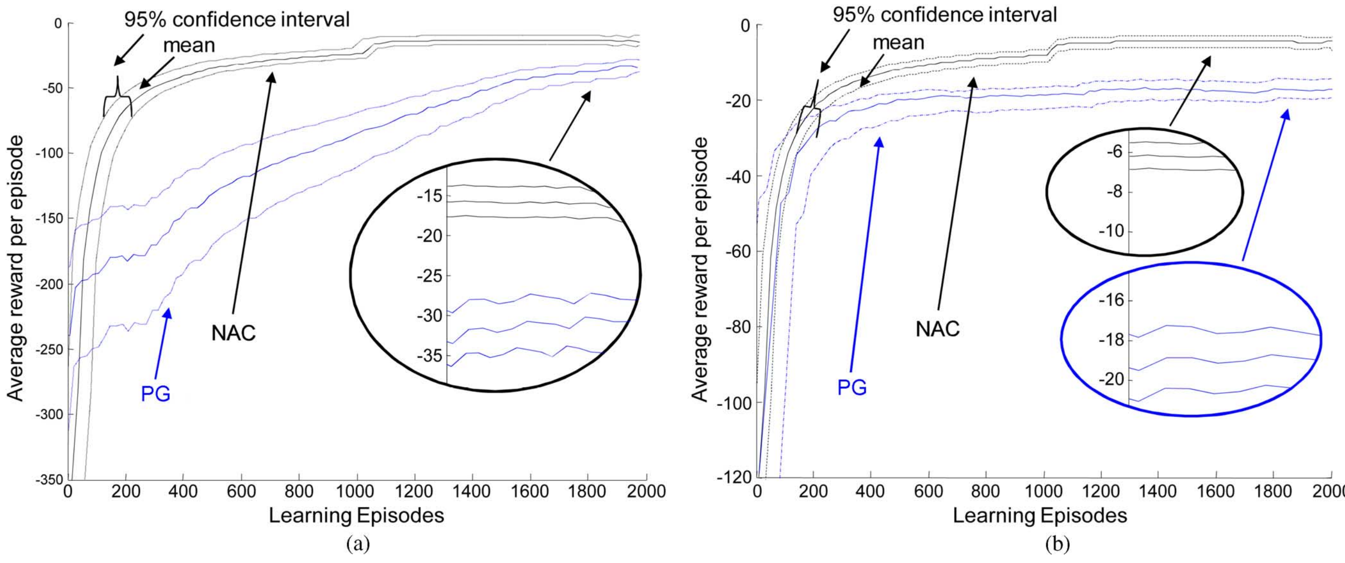


Fig. 17. (a) Learning evolution for the sway policy. Comparison with respect to the number of episodes between the NAC and basic PG algorithms. Results averaged over 100 independent runs. (b) Learning evolution for the yaw policy. Comparison with respect to the number of episodes between the NAC and basic PG algorithms. Results averaged over 100 independent runs.

curve obtained under the same conditions with a basic PG algorithm is also depicted. The 95% confidence intervals for both curves are shown. After running 100 independent runs with the NAC algorithm, the average reward per episode once the optimal sway policy is learned is set around -15, notably better results than the ones obtained with the basic PG, which received their best mark at -35. The average number of episodes needed by the NAC to learn this sway optimal policy is approximately 11×10^2 episodes, while the basic PG needs twice the number of episodes to obtain its best result, which is still worse than the best one obtained by the NAC.

Fig. 17(b) shows the same learning comparison, but this time for the learning evolution of the yaw policy. The average reward per episode obtained by the AC algorithm once the optimal sway policy is learned is set around -6. Again these results are better than the ones obtained with the PG, which received their best mark at -19. The average number of episodes needed by the NAC to learn this yaw optimal policy is approximately 14×10^2 episodes, while the basic PG reaches average reward values close to its best mark at relatively early episodes 6×10^2. Once the learning process is considered finished, resultant policies obtained with the NAC algorithm with its correspondent parameters are transferred to the underwater robot *Ictineu* AUV, ready to be tested in a real environment.

B. Phase Two: Real Learning Results for the Programmed RL Behavior cableTracking

The transferred policies are ready to start the real learning on the robot. The main objective of the online real learning step is to improve the learned-in-simulation policies while interacting with the real environment. Fig. 18(a) shows the evolution of the state variable Θ along different learning runs. The first run, depicted with black color, illustrates the performance of the Θ angle after 90 s of online learning. Each run lasts approximately 90 s, which corresponds to the time needed by the robot to make a full run from one corner of the pool to the opposite, navigating around 9 m of cable. Once the robot reaches the corner of the pool, it automatically disconnects the learning behavior and enables another behavior which makes the robot turn 180° around the Z-axis, facing the cable in the opposite direction. Then, it enables the learning behavior again and a new run starts. Fig. 18(a) shows how, as the learning process goes on, policies improve significantly. After 20 trials (around 1800 s from the start) the blue line shows a smoother Θ evolution along the run, denoting a significant improvement of the trajectories. Fig. 18(b) depicts the results regarding the state variable x_g along the same learning runs.

Fig. 19 shows the accumulated reward evolution during the real learning phase. This figure is the result of adding the total accumulated reward per trials of both policies, sway and yaw, over five independent tests of 40 runs each, remembering that each run lasts approximately 90 s, and represents a full length run from one side of the cable to the other, navigating along 9 m. It can be observed that in all five tests, the algorithm converges to a better policy as shown by the increasing evolution of the averaged rewards.

C. Phase Three: Simulated Execution of the Whole Underwater Cable Tracking Mission

Since the primitives have been tested with the *Ictineu* AUV in a water pool, the whole proposed mission has been simulated using a hardware-in-the-loop simulator [94]. The simulated environment allows us to control all the events produced during the mission execution and also gives us the opportunity to induce some errors, task failures, and alarms. Thus, it is easier to check if the vehicle reacts as specified in the mission plan. Similar executions have been obtained by the offline and onboard plans. The main difference is the time spent by the onboard planner when computing a new plan (a few seconds), in which the vehicle keeps its position until a new plan is generated. It is worth noting that 12 plans have been generated by the

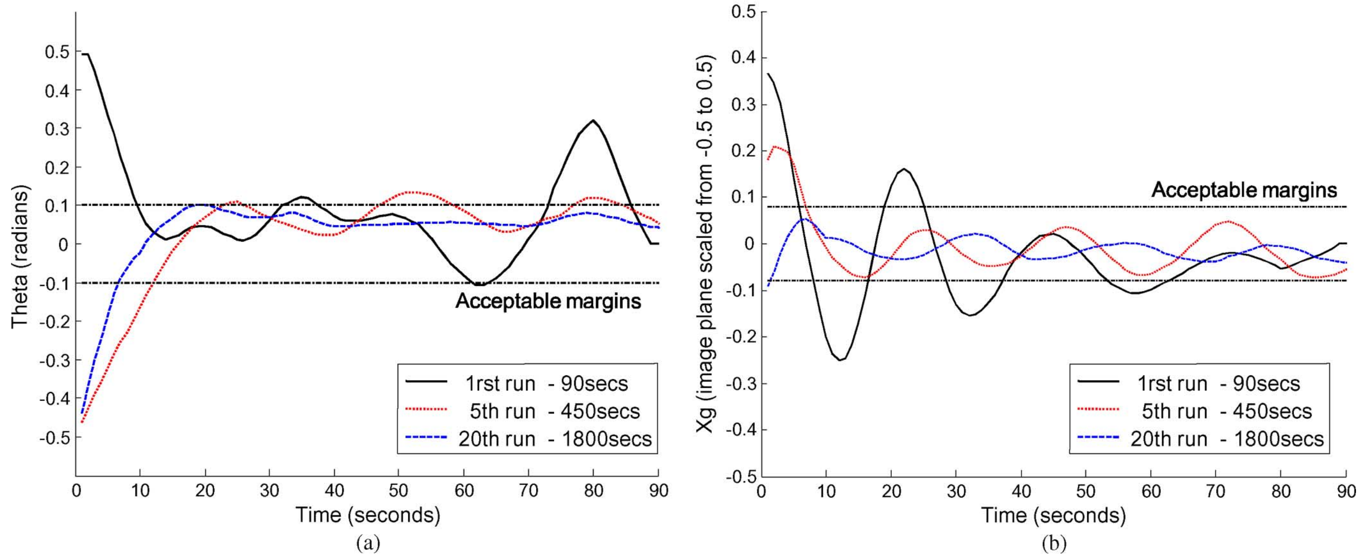


Fig. 18. Policy improvement through different runs of real learning. Evolution of Θ angle (a) and centroid x_g (b) performance.

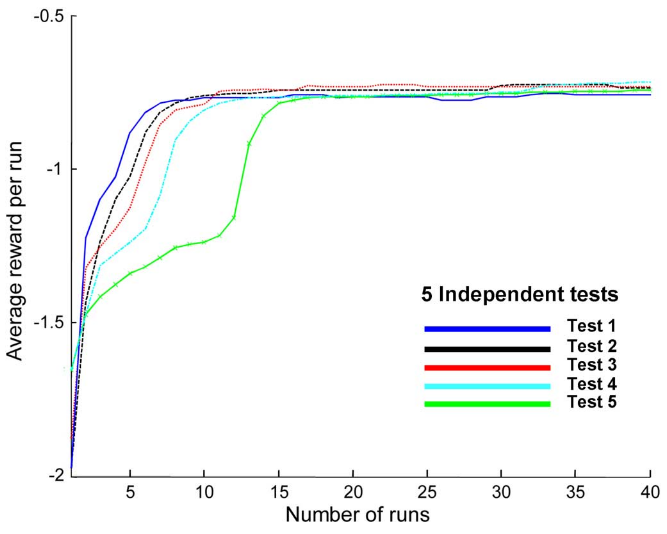


Fig. 19. Accumulated reward progression for five independent tests. Reward shown is the result of adding the accumulated reward for both sway and yaw policies.

onboard planner to complete the mission shown in Fig. 20(a). However, due to the simplicity of the mission, a few seconds have been required to compute all these plans online. Fig. 20(a) shows the trajectory performed by the vehicle during the execution of the proposed mission. The AUV is submerged and localizes the cable [Fig. 20(b)]. Then, it tracks the underwater cable for more than 130 m using the *cableTracking* primitive specially develop by this purpose. The RL algorithm that implement this primitive is constantly updating the stimulus-to-action mapping to improve its performance. Each time that the cable is not visible by the vision system (i.e., because the cable is buried by sand) a search procedure based on the last known bearing is enabled to find it [Fig. 20(c)]. Moreover, if the vehicle's navigation accuracy is below a threshold, the vehicle surfaces to take a position fix [Fig. 20(d)]. The mission finalizes if after several time searches for the cable it is impossible to find it [Fig. 20(e)] or if any alarm rises (e.g., the vehicle runs out of battery).

VIII. Conclusion

In this paper, COLA2, a control architecture for an AUV, has been presented. The proposal implements a component oriented layer-based control architecture structured in three layers: the reactive layer, the execution layer, and the mission layer. Concerning the reactive layer, to improve the vehicle primitives' adaptability to unknown changing environments, RL techniques have been programmed. Starting from a learned-in-simulation policy, the RL-based primitive *cableTracking* has been trained to follow an underwater cable in a real environment inside a water tank using the *Ictineu* AUV. The execution layer implements a DES based on PNs. PNs have been used to safely model the primitives' execution flow by means of PNBBs. Task PNBBs have been used to supervise the execution of the primitives while control structure PNBBs have been used to compose tasks to describe their execution flow. All these PNBBs have been designed according to some reachability properties showing that it is possible to compose them preserving these qualities. The mission layer describes the mission phases by means of a high-level MCL, which is automatically compiled into a PN. The MCL presents agreeable properties of simplicity and structured programming. Also, it offers the possibility of sequential/parallel, conditional, and iterative task execution. MCL can be used to describe an offline imperative mission to be executed by the vehicle. First, this mission is compiled into a PN. Then, the Petri net player (PNP) is the component which executes this PN in real time. This component communicates with the reactive layer through an AAC keeping the execution and mission layers independent from the hardware being used. Additionally to predefined missions, it is possible to describe planning operators, in charge of solving a particular phase of a mission. Thus, an onboard planner will be able to sequence the available operators to achieve the proposed goals. The whole

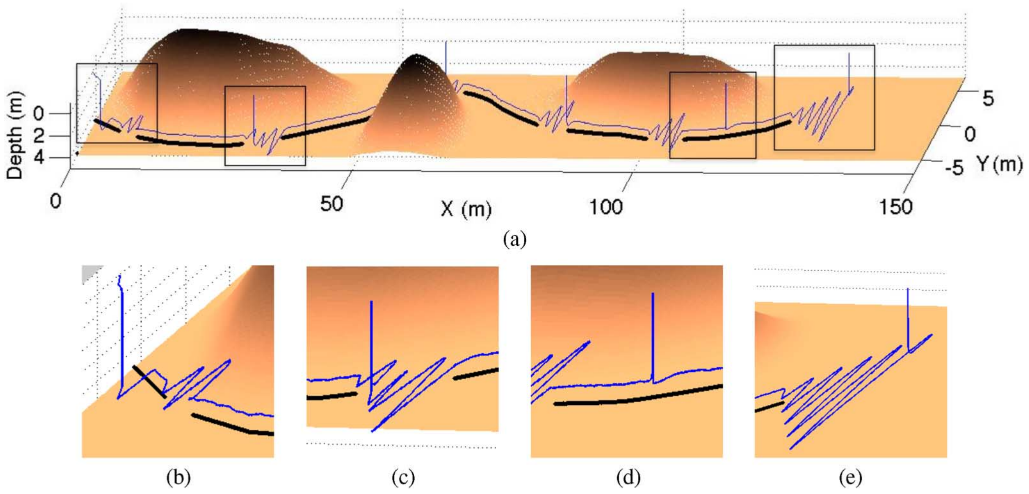


Fig. 20. Simulated mission results. The image represents the environment with the partially buried cable shown in black while the obtained vehicle's trajectory after executing the cable tracking mission is shown in blue.

architecture has been validated in a cable tracking mission divided in two main phases. In the first phase, the *cableTracking* primitive of the reactive layer has been trained to follow a cable in a water tank with the vehicle *Ictineu* AUV, one of the research platforms available in the VICOROB. In the second phase, the whole architecture has been proved in a realistic simulation of a whole cable tracking mission.

This work has accomplished several proposed goals. Moved by the considerable interest that, in the past few years, has arisen around AUV applications, this paper demonstrates the feasibility of using learning algorithms in the reactive layer to help AUVs perform autonomous tasks. The adaptive RL programmed primitive *cableTracking* has demonstrated good results. The *Ictineu* AUV has learned to perform a real-visual-based cable tracking task in a two-step learning process. First, a policy has been computed by means of simulation where a hydrodynamic model of the vehicle simulates the cable following the task. Second, once the simulated results were good enough, the learned-in-simulation policy has been transferred to the vehicle where the learning procedure continued online in a water tank. A PG-based algorithm, the NAC, was selected to solve the problem through online interaction with the environment in both steps: the simulation and the real learning. For performance comparison purposes, the results obtained with the NAC algorithm during the simulated phase have been compared with a basic PG algorithm tested in the same conditions. The comparative results reveal that the NAC algorithm outperforms a basic gradient method in both convergence speed and final performance of the learned policy.

Several innovations have been also included in the execution and mission layers. First, a new method to define a set of building blocks using PN that shows how these blocks have to be placed to create a larger structure is presented. These PNBBs are conditioned to several constraints. On the one hand, they have to share a common interface that determines the number of actions/events to be received/sent for each task. On the other hand, a reachability analysis has to be performed to ensure that each block evolves free of deadlocks from a valid initial state to a valid final state. Then, it is possible to describe a large structure by means of composing a PNBB in which the checked properties hold. In addition, to allow AUV users to easily define a mission composing PNBBs, a completely new language has been designed and implemented. The MCL is a high-level imperative language that automatically compiles the mission program into a formal PN, avoiding the tedious part of programming a mission directly manipulating the PN. Finally, to increase the deliberative properties of COLA2, an automated planning algorithm has been proposed. It relies on a world modeler that transforms low-level perceptions into high-level symbols used to plan. Moreover, MCL submissions have been used as planning operators to ensure a predictable behavior while simplifying the whole system.

As future work, the whole architecture should be tested in real scenarios and in more different underwater tasks. Operating in real scenarios like coastal waters will introduce new challenges: unpredictable water currents, irregular seafloor, and changing light conditions are only some of them. One of the objectives of this paper was to demonstrate the feasibility of RL techniques to learn autonomous underwater tasks. Since the complexity of the cable tracking task presented in the final experiments did not represent a problem for algorithms like NAC, the development of more complex tasks may offer to these kind of algorithms the opportunity to prove themselves in harder underwater operations. Also, the deliberative algorithms implemented in the mission layer should be improved adding, for instance, some heuristics, and tested in these same scenarios.

REFERENCES

[1] R. Arkin, *Behavior-Based Robotics*. Cambridge, MA: MIT Press, 1998, ch. 4.

[2] D. Kortenkamp and R. Simmons, "Robotic systems architectures and programming," in *Handbook of Robotics*. New York: Springer-Verlag, 2008, pp. 187–206.

[3] M. Saptharishi, C. S. Oliver, C. Diehl, K. Bhat, J. Dolan, A. Trebi-Ollennu, and P. Khosla, "Distributed surveillance and reconnaissance using multiple autonomous ATVs: Cyberscout," *IEEE Trans. Robot. Autom.*, vol. 18, no. 5, pp. 826–836, Oct. 2002.

[4] M. Beetz, T. Arbuckle, T. Belker, A. Cremers, D. Schulz, M. Bennewitz, W. Burgard, D. Hahnel, D. Fox, and H. Grosskreutz, "Integrated, plan-based control of autonomous robot in human environments," *IEEE Intell. Syst.*, vol. 16, no. 5, pp. 56–65, Sep.-Oct. 2001.

[5] W. Burgard, A. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, "Experiences with an interactive museum tour-guide robot," *Artif. Intell.*, vol. 114, no. 1–2, pp. 3–55, 1999.

[6] R. M. Turner, "Intelligent mission planning and control of autonomous underwater vehicles," in *Proc. Int. Conf. Autom. Planning Scheduling*, Monterey, CA, Jun. 5–10, 2005, pp. 5–8.

[7] P. Ridao, J. Batlle, and M. Carreras, " $o2ca2$, a new object oriented control architecture for autonomy: The reactive layer," *Control Eng. Practice*, vol. 10, no. 8, pp. 857–873, 2002.

[8] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[9] R. Sutton and A. Barto*, Reinforcement Learning, An Introduction*. Cambridge, MA: MIT Press, 1998, ch. 1.

[10] D. Ribas, N. Palomeras, P. Ridao, M. Carreras, and Hernandez, "ICTINEU AUV wins the first SAUC-E competition," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2007, pp. 151–156.

[11] N. Nilsson*, Principles of Artificial Intelligence*. Wellsboro, PA: Tioga Publishing Company, 1980, ch. 7.

[12] R. Fikes and N. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, pp. 189–208, 1971.

[13] N. Nilsson, "Shakey the robot," SRI International, Menlo Park, CA, Tech. Rep. 323, 1984.

[14] J. Albus, "Outline for a theory of intelligence," *IEEE Trans. Syst. Man Cybern.*, vol. 21, no. 3, pp. 473–509, May–Jun. 1991.

[15] H. Huang, "An architecture and a methodology for intelligent control," *IEEE Expert, Intell. Syst. Appl.*, vol. 11, no. 2, pp. 46–55, Apr. 1996.

[16] D. Lefebvre and G. Saridis, "A computer architecture for intelligent machines," in *Proc. IEEE Int. Conf. Robot. Autom.*, Nice, France, 1992, pp. 245–250.

[17] R. Chatila and J. Laumond, "Position referencing and consistent world modelling for mobile robots," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1985, pp. 138–170.

[18] J. Laird and P. Rosenbloom, "Integrating, execution, planning, and learning in Soar for external environments," in *Proc. 8th Annu. Meeting Amer. Assoc. Artif. Intell.*, T. S. W. Dietterich, Ed., Jul.–Aug. 1990, pp. 1022–1029.

[19] R. Brooks, "A robust layered control system for a mobile robot," *IEEE J. Robot. Autom.*, vol. 2, no. 1, pp. 14–23, Apr. 1986.

[20] S. Rosenschein and L. Kaelbling, "The synthesis of digital machines with provable epistemic properties," in *Proc. Conf. Theor. Aspects Reason. About Knowl.*, 1986, pp. 83–98.

[21] P. Agre and D. Chapman, "Pengi: An implementation of a theory of activity," in *Proc. 6th Annu. Meeting Amer. Assoc. Artif. Intell.*, Seattle, WA, 1987, pp. 268–272.

[22] J. Connell, "SSS: A hybrid architecture applied to robot navigation," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1992, pp. 2719–2724.

[23] I. Horswill, "Polly: A vision-based artificial agent," in *Proc. IEEE Nat. Conf. Artif. Intell.*, 1993, pp. 824–829.

[24] R. Arkin, "Motor schema-based mobile robot navigation," *Int. J. Robot. Res.*, vol. 8, no. 4, pp. 92–112, Aug. 1989.

[25] R. Firby, "Adaptive execution in complex dynamic worlds," Ph.D. dissertation, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1989.

[26] R. Arkin and T. Balch, "Aura: Principles and practice in review," *J. Exp. Theor. Artif. Intell.*, vol. 9, pp. 175–189, 1997.

[27] D. Lyons, "Planning, reactive," in *Encyclopedia of Artificial Intelligence*, S. Shapiro, Ed., 2nd ed. New York: Wiley, 1992, pp. 1171–1182.

[28] E. Gat, "Reliable goal-directed reactive control for real-world autonomous mobile robots," Ph.D. dissertation, Dept. Comput. Sci. Res., Virginia Polytechnic State Univ., Blacksburg, VA, 1991.

[29] W. Garage, "Robot operating system," 2010 [Online]. Available: http://www.ros.org

[30] G. Biggs, T. Collett, B. Gerkey, A. Howard, N. Koenig, J. Polo, R. Rusu, and R. Vaughan, "Player and stage," 2010 [Online]. Available: http://playerstage.sourceforge.net/

[31] P. M. Newman, "MOOS—Mission Orientated Operating Suite," 2005.

[32] D. Ribas, P. Ridao, and J. Neira*, Underwater SLAM for Structured Environments Using an Imaging Sonar*, ser. Springer Tracts in Advanced Robotics. New York: Springer-Verlag, 2010, pp. 59–64, Sec. 5.3.

[33] M. Carreras, J. Batlle, and P. Ridao, "Hybrid coordination of reinforcement learning-based behaviors for AUV control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2001, vol. 3, pp. 1410–1415.

[34] W. Smart and L. Kaelbling, "Practical reinforcement learning in continuous spaces," in *Proc. Int. Conf. Mach. Learn.*, 2000, pp. 903–910.

[35] N. Hernandez and S. Mahadevan, "Hierarchical memory-based reinforcement learning," in *Advances in Neural Information Processing Systems (NIPS)*. Cambridge, MA: MIT Press, 2000, pp. 1047–1053.

[36] M. Carreras, P. Ridao, and A. El-Fakdi, "Semi-online neural-Q-learning for real-time robot learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Las Vegas, NV, Oct. 27–31, 2003, pp. 662–667.

[37] J. Peters, "Machine learning of motor skills for robotics," Ph.D. dissertation, Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, 2007.

[38] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Beijing, China, Oct. 9–15, 2006, pp. 2219–2225.

[39] R. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems (NIPS)*. Cambridge, MA: MIT Press, 2000, vol. 12, pp. 1057–1063.

[40] S. Singh, T. Jaakkola, and M. Jordan, "Learning without state-estimation in partially observable Markovian decision processes," in *Proc. 11th Int. Conf. Mach. Learn.*, 1994, pp. 284–292.

[41] C. Anderson, "Approximating a policy can be easier than approximating a value function," Colorado State Univ. , Fort Collins, CO, Comput. Sci. Tech. Rep., 2000.

[42] J. Baxter and P. Bartlett, "Direct gradient-based reinforcement learning," in *Proc. Int. Symp. Circuits Syst.*, Geneva, Switzerland, May 2000, vol. 3, pp. 271–274.

[43] C. Watkins and P. Dayan, "Q-learning," *J. Mach. Learn.*, vol. 8, pp. 279–292, 1992.

[44] H. Benbrahim and J. Franklin, "Biped dynamic walking using reinforcement," *Robot. Autonom. Syst.*, vol. 22, pp. 283–302, 1997.

[45] V. Gullapalli, J. J. Franklin, and H. Benbrahim, "Acquiring robot skills via reinforcement learning," *IEEE Control Syst.*, vol. 4, Special Issue on Robotics: Capturing Natural Motion, no. 1, pp. 13–24, Feb. 1994.

[46] H. Benbrahim, J. Doleac, J. Franklin, and O. Selfridge, "Real-time learning: A ball on a beam," in *Proc. Int. Joint Conf. Neural Netw.*, Baltimore, MD, 1992, pp. 98–103.

[47] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *Proc. Eur. Conf. Mach. Learn.*, 2005, pp. 280–291.

[48] D. Aberdeen, "Policy-gradient algorithms for partially observable Markov decision processes," Ph.D. dissertation, Comput. Sci. Lab., Australian Nat. Univ., Canberra, A.C.T., Australia, Apr. 2003.

[49] P. Maes and R. Brooks, "Learning to coordinate behaviors," in *Proc. 8th Meeting Amer. Assoc. Artif. Intell.*, 1990, pp. 796–802.

[50] D. Gachet, M. Salichs, L. Moreno, and J. Pimental, "Learning emergent tasks for an autonomous mobile robot," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Munich, Germany, Sep. 1994, pp. 290–297.

[51] Z. Kalmar, C. Szepesvari, and A. Lorincz, "Module-based reinforcement learning: Experiments with a real robot," *J. Autonom. Robots*, vol. 5, no. 3–4, pp. 273–295, Aug. 1998.

[52] E. Martinson, A. Stoytchev, and R. Arkin, "Robot behavioral selection using Q-learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Lausanne, Switzerland, 2002, pp. 970–977.

[53] M. Ryan and M. Pendrith, "RL-TOPs: An architecture for modularity and re-use in reinforcement learning," in *Proc. 15th Int. Conf. Mach. Learn.*, Madison, WI, 1998, pp. 481–487.

[54] C. Touzet, "Neural reinforcement learning for behavior synthesis," *Robot. Autonom. Syst.*, vol. 22, pp. 251–281, 1997.

[55] Y. Takahashi and M. Asada, "Vision-guided behavior acquisition of a mobile robot by multi-layered reinforcement learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2000, vol. 1, pp. 395–402.

[56] J. Shackleton and M. Gini, "Measuring the effectiveness of reinforcement learning for behavior-based robotics," *Adaptive Behav.*, vol. 5, no. 3/4, pp. 365–390, 1997.

[57] S. Mahadevan and J. Connell, "Automatic programming of behavior-based robots using reinforcement learning," *Artif. Intell.*, vol. 55, pp. 311–365, 1992.

[58] A. El-Fakdi, M. Carreras, and P. Ridao, "Towards direct policy search reinforcement learning for robot control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2006, pp. 3178–3183.

[59] A. El-Fakdi and M. Carreras, "Policy gradient based reinforcement learning for real autonomous underwater cable tracking," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2008, pp. 3635–3640.

[60] J. Boyan, "Least-squares temporal difference learning," in *Proc. 16th Int. Conf. Mach. Learn.*, 1999, pp. 49–56.

[61] R. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *J. Mach. Learn.*, vol. 8, pp. 229–256, 1992.

[62] M. Barbier, J. Lemaire, and N. Toumelin, "Procedures planner for an AUV," presented at the 12th Int. Symp. Unmanned Untethered Submersible Technol., Durham, NH, Aug. 27–29, 2001.

[63] A. Healey, D. Marco, P. Oliveira, and A. Pascoal, "Strategic level mission control—An evaluation of CORAL and PROLOG implementations for mission control specifications," in *Proc. Symp. Autonom. Underwater Veh. Technol.*, 1996, pp. 125–132.

[64] F. Wang, K. Kyriakopoulos, and Tsolkas, "A Petri-net coordination model for an intelligent mobile robot," *IEEE Trans. Syst. Man Cybern.*, vol. 21, no. 4, pp. 777–789, 1991.

[65] P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre, "Mission control of the Marius AUV: System design, implementation, and sea trials," *Int. J. Syst. Sci.*, vol. 29, no. 10, pp. 1065–1080, 1998.

[66] M. Caccia, P. Coletta, G. Bruzzone, and G. Veruggio, "Execution control of robotic tasks: A Petri net-based approach," *Control Eng. Practice*, vol. 13, no. 8, pp. 959–971, 2005.

[67] V. Ziparo and L. Iocchi, "Petri net plans," presented at the ATPN/ ACSD 4th Int. Workshop Model. Objects Compon. Agents, Turku, Finland, Jun. 26, 2006.

[68] H. Costelha and P. Lima, "Modelling, analysis and execution of robotic tasks using Petri nets," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2007, pp. 1449–1454.

[69] C. Petri, "Kommunikation mit automaten," Schriften des Institutes für Instrumentelle Mathematik, Bonn, Germany, Jan. 1962.

[70] N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre, "Towards a mission control language for AUVs," in *Proc. 17th Int. Fed. Autom. Control World Congr.*, 2008, pp. 15 028–15 033.

[71] J. Evans, C. Sotzing, P. Patron, and D. Lane, "Cooperative planning architectures for multi-vehicle autonomous operations," Ocean Syst. Lab., Heriot-Watt Univ., Edinburgh, U.K., Tech. Rep., Jul. 2006.

[72] P. Patrón, E. Miguelañez, Y. R. Petillot, D. M. Lane, and J. Salvi, "Adaptive mission plan diagnosis and repair for fault recovery in autonomous underwater vehicles," in *Proc. IEEE OCEANS Conf.*, Sep. 2008, DOI: 10.1109/OCEANS.2008.5151975.

[73] K. Rajan, C. McGann, F. Py, and H. Thomas, "Robust mission planning using deliberative autonomy for autonomous underwater vehicles," in *Proc. ICRA Workshop Robot. Challenging Hazardous Environ.*, 2007, pp. 21–25.

[74] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt, "Idea: Planning at the core of autonomous reactive agents," presented at the 3rd Int. NASA Workshop Planning Scheduling for Space, Houston, TX, Oct. 27–29, 2002.

[75] F. Boussinot and R. de Simone, "The ESTEREL language," *Proc. IEEE*, vol. 79, no. 9, pp. 1293–1304, Sep. 1991.

[76] T. Kim and J. Yuh, "Task description language for underwater robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Sep. 2003, vol. 1, pp. 565–570.

[77] R. Firby, "An investigation into reactive planning in complex domains," in *Proc. 6th Nat. Conf. Artif. Intell.*, 1987, pp. 202–206.

[78] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves, "The Petri net markup language and ISO/IEC 15909-2," in *Proc. 10th Int. Workshop Practical Use of Colored Petri Nets and the CPN Tools*, 2009, pp. 101–120.

[79] N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre, "Using Petri nets to specify and execute missions for autonomous underwater vehicles," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2009, pp. 4439–4444.

[80] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning*. San Mateo, CA: Morgan Kaufmann, 2004, ch. 1–5.

[81] I.-B. Jeong and J.-H. Kim, "Multi-layered architecture of middleware for ubiquitous robot," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, 2008, pp. 3479–3484.

[82] J. Amat, J. Batlle, A. Casals, and J. Forest, "GARBI: A low cost ROV, constrains and solutions," presented at the 6ème Seminaire IARP en robotique sous-marine, Toulon-La Seyne, France, 1996.

[83] J. Batlle, P. Ridao, R. Garcia, M. Carreras, X. Cufi, A. El-Fakdi, D. Ribas, T. Nicosevici, and E. Batlle, *URIS: Underwater Robotic Intelligent System*, 1st ed. Madrid, Spain: Instituto de Automatica Industrial, Consejo Superior de Investigaciones Cientificas, 2004, ch. 11, pp. 177–203.

[84] D. Ribas, N. Palomeras, P. Ridao, M. Carreras, and E. Hernandez, "Ictineu AUV wins the first SAUC-E competition," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2007, pp. 151–156.

[85] Y. Ito, N. Kato, J. Kojima, S. Takagi, K. Asakawa, and Y. Shirasaki, "Cable tracking for autonomous underwater vehicle," in *Proc. IEEE Symp. Autonom. Underwater Veh. Technol.*, 1994, pp. 218–224.

[86] K. Asakawa, J. Kojima, Y. Kato, S. Matsumoto, N. Kato, T. Asai, and T. Iso, "Design concept and experimental results of the autonomous underwater vehicle Aqua Explorer 2 for the inspection of underwater cables," *Adv. Robot.*, vol. 16, no. 1, pp. 27–42, 2002.

[87] J. Evans, Y. Petillot, P. Redmond, M. Wilson, and D. Lane, "AUTOTRACKER: AUV embedded control architecture for autonomous pipeline and cable tracking," in *Proc. MTS/IEEE OCEANS Conf.*, San Diego, CA, Sep. 2003, pp. 2651–2658.

[88] M. Iwanowski, "Surveillance unmanned underwater vehicle," in *Proc. IEEE OCEANS Conf.*, 1994, pp. 1116–1119.

[89] A. Balasuriya and T. Ura, "Vision based underwater cable detection and following using AUVs," in *Proc. MTS/IEEE OCEANS Conf.*, Biloxi, MS, Oct. 2002, vol. 3, pp. 1582–1587.

[90] A. Ortiz, J. Antich, and B. Oliver, "A particle filter-based approach for tracking undersea narrow telecommunication cables," *Int. J. Mach. Vis. Appl.*, vol. 22, no. 2, pp. 283–302, Mar. 2011.

[91] R. Garcia, X. Cufí, and J. Batlle, "Detection of matchings in a sequence of underwater images through texture analysis," in *Proc. Int. Conf. Image Process.*, 2001, vol. 1, pp. 361–364.

[92] A. Healey, *Guidance Laws, Obstacle Avoidance and Artificial Potential Functions*, ser. Advances in Unmanned Marine Vehicles. London, U.K.: IEE, 2006, pp. 43–66.

[93] A. El-Fakdi, M. Carreras, and E. Galceran, "Two steps natural actor critic learning for underwater cable tracking," in *Proc. IEEE Int. Conf. Robot. Autom.*, Anchorage, AK, May 2010, pp. 2267–2272.

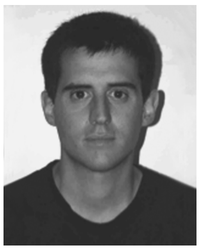
[94] P. Ridao, A. Tiano, A. El-Fakdi, M. Carreras, and A. Zirilli, "On the identification of non-linear models of unmanned underwater vehicles," *Control Eng. Practice*, vol. 12, pp. 1483–1499, 2004.

Narcis Palomeras received the Ms.C. degree in computer science from the University of Girona, Girona, Spain, in 2005, where he is currently working toward the Ph.D. degree in information technologies.

His research interests are focused on autonomous control architectures, specifically in developing a mission control system (MCS) for an autonomous underwater vehicle (AUV) based on Petri nets. He is involved in national projects and European research networks about underwater robotics and is a member of the Research Center in Underwater Robotics (CIRS) of Girona.

Andres El-Fakdi received the Ms.C. degree in industrial engineering from the University of Girona, Girona, Spain, in 2003, where he is currently working toward the Ph.D. degree in information technologies.

His research interests are focused on reinforcement learning techniques, specifically in developing policy gradient algorithms to learn different behaviors for autonomous underwater vehicles (AUVs). He is involved in national projects and European research networks about underwater robotics and is a member of the Research Center in Underwater Robotics (CIRS) of Girona.



Marc Carreras received the M.Sc. degree in industrial engineering and the Ph.D. degree in computer engineering from the University of Girona, Girona, Spain, in 1998 and 2003, respectively.

His research activity is mainly focused on robot learning and intelligent control architectures of autonomous underwater vehicles (AUVs). He joined the Institute of Informatics and Applications, University of Girona, in September 1998. Currently, he is an Associate Professor with the Department of Computer Engineering of the University of Girona and a member of the Research Center in Underwater Robotics (CIRS) of Girona. He is involved in national and European research projects and networks about underwater robotics.

Pere Ridao received the Ms.C. degree in computer science from the Technical University of Catalonia, Barcelona, Spain, in 1993 and the Ph.D. degree in computer engineering from the University of Girona, Girona, Spain, in 2001.

His research activity is mainly focused on underwater robotics in research topics such as intelligent control architectures, unmanned underwater vehicle (UUV) modeling and identification, simulation, navigation, mission control, and real-time systems. Currently, he is an Associate Professor with the Department of Computer Engineering, University of Girona and the Head of the Research Center in Underwater Robotics (CIRS) of Girona. He is involved in national projects and European research networks about underwater robotics and some technology transference projects about real-time and embedded systems.

Dr. Ridao is member of the International Federation of Automatic Control (IFAC) Technical Committee on Marine Systems, member of the editorial board of Springer's *Intelligent Service Robotics* journal, secretary of the IEEE Oceanic Engineering Society (OES) Spanish chapter, and also a board member of the IEEE Robotics and Automation Society (RAS) Spanish chapter.