# ESLAS – a robust layered learning framework

Willi Richert

Riccardo Tornese

*Abstract*— **With increasing capabilities today robots get more and more complex to program. Not only the low-level skills and different strategies for different subgoals have to be specified, which by itself is not a trivial task even for simple domains. Both, the skill set and the strategies, also have to be compatible with each other. This turns out to be a major hassle as they are designed and implemented under assumptions about the future environment and conditions the robot will be faced with, that usually do not hold in reality.**

**The *Evolving Societies of Learning Autonomous Systems* architecture (ESLAS) is targeted to this problem. With minimal need for specification, it is able to learn skills and strategies independently in order to accomplish different goals, which the designer can specify by means of an intuitive motivation system. In addition, it is able to handle system and environmental changes by learning autonomously at the different levels of abstraction. It is achieving this in continuous and noisy environments by 1) an active strategy-learning module that uses reinforcement learning and 2) a dynamically adapting skill module that proactively explores the robot's own action capabilities and thereby provides actions to the strategy module. We demonstrate the feasibility of simultaneously learning low-level skills and high-level strategies in a Capture-The-Flag scenario. Thereby, the robot drastically increases its overall autonomy.**

*Index Terms*— **autonomous framework, strategy learning, skill learning, robotics**

## I. Introduction

Whenever a robot has to be programmed, its designer has to make many assumptions about the future environment to keep the task tractable. Even more so, if the behavior that the robot will have to exhibit is so complex that it needs different levels of abstractions. The assumptions typically decrease the robot's autonomy and robustness in later application. A learning robot architecture is therefore desirable that places a minimum of assumptions into its algorithms in order to increase its robustness and autonomy. This architecture should combine top-down goal specification with bottom-up exploration of its own capabilities. The desired characteristics of such and architecture are the following:

- The ability to learn and apply continuous actions (skills) in noisy domains.
  - The skill learner should find out by itself what types of capabilities actually are learnable before it starts trying to learn specific skills.
  - Skills should be able to be adapted while being executed.
  - The ability to find a "good enough" action to be executed "fast enough".
  - Skills should provide enough data, methods, and means to categorize observed performance of other robots in order to learn from them for increased learning speed.
  - Skills should abstracted from the strategy and be visible to it only by some kind of *handle*.
- The capability to find state abstractions that are able to distinguish between sufficiently distinct states from the view of the learned skill set while maintaining good generalization.
  - It should account for continuous time.
  - It should support multiple possibly contradicting goals.
  - It should be able to learn from delayed feedback from the environment.

This naturally leads to an architecture consisting of three layers: the motivation, the strategy, and the skill layer. The overall goal can be specified intuitively by different drives that make up the robot's motivation layer. Each drive is representing one sub-goal. The strategy layer has the task to group the infinitely large state space into a small number of abstract regions in order to escape the curse of dimensionality and determine the optimal action for each one of those sub-goals. As the environment can change during runtime, the strategy layer also has to maintain a model about its behavior in that environment. The low-level skills that make up the overall behavior is the task of the skill layer. It has to find out, which actions the robot is actually capable of. It is in charge not only of exploring its own capabilities, but also to optimize them while normally executing them.

Our approach is thus providing a framework that combines strategy learning with Developmental Robotics principles to satisfy the different sub-goals of the overall motivation. In this article, we present a significantly extended version of our previous work [1]. The strategy layer is more autonomous and robust to environmental change in that it does not rely anymore on standard model-based Reinforcement Learning. Instead, it deploys intertwined state abstraction with model-based Reinforcement Learning. The skills are not anymore restricted to fixed models. Instead we fully revised our skill system that allows now for arbitrary models and includes developmental robotics principles in that it is able to learn what is actually able to learn [2], [3].

This work is partially supported by the DFG-Schwerpunktprogramm 1183: Organic Computing

Willi Richert is with Faculty of of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany, Fürstenallee 11, 33102 Paderborn, Germany richert@c-lab.de, Riccardo Tornese is graduate student of Information Technology Engineering of Politecnico di Milano, piazza Leonardo Da Vinci 32, 20133 Milan, Italy riccardo.tornese@mail.polimi.it

## II. Related work

When ignoring the need for action recognition, which is necessary for perception-based imitation and coordination, there seems to be a lot of research done in the area of continuous state and action spaces.

### A. Model-free approaches

Hasselt and Wiering devised the *Continuous Actor Critic Learning Automaton* approach, which empowers reinforcement learning to operate on continuous state and action spaces [4]. They calculate real valued actions by interpolating the available discrete actions based on their utility values. Therefore, the performance is highly dependent on initial assumptions about the value function.

It is obvious that a full search in continuous state and action spaces is infeasible. For reinforcement learning approaches to be applied in realistic domains, it is therefore vital to limit the search to small areas in the search space. One approach to do that is the *Actor-Critic* method [5]. It separates the presentation of the policy from the value function. The actor maintains for each state a probability distribution over the action space. The critic is responsible for providing they reward from the actions taken by the actor, which in turns modifies its policy. As this relieves the designer from assumptions about the value function, it introduces new assumptions about the underlying probability distribution. To overcome this problem Lazaric et al. devised *Sequential Monte Carlo Learning* [6], which combines the actor critic method with a nonparametric representation of the actions. After initially being drawn from a prior distribution, they are resampled dependent on the utility values learned by the critic.

Bonarini et al. developed *Learning Entities Adaptive Partitioning* (LEAP) [7], a model-free learning algorithm that uses overlapping partitions, which are dynamically modified to learn near-optimal policies with a small number of parameters. Whenever it detects incoherence between the current action values and the actual rewards from the environment it modifies those partitions. In addition, it is able to prune over-refined partitions. Thereby it creates a multi-resolution state representation specialized only where it is actually needed. The action space is not considered by this approach. In their grid world experiment, they use a fixed set of predefined actions.

### B. Model-based approaches

The *Adaptive Modelling and Planning System* (AMPS) by Kochenderfer [8] maintains an adaptive representation of both the state and the action space. In his approach, the abstraction of the state and action space is combined with policy learning in a smart way: states are grouped into abstract regions, which have the common property that perception-action-traces, previously performed in that region, "feel" similar in terms of failure rates, duration, and expected reward. It does so by splitting and merging abstract states at runtime. AMPS not only dynamically abstracts the state space into regions, but also the action space into action regions. This is, however, done in a very artificial way that could not yet been shown to work in real world domains.

Although our strategy layer is inspired by AMPS, we differ from it in the following important points: AMPS applies the splitting and merging also to the action space, which works fine in artificial domains but will not cope with the domain dependency one is typically faced with in real environments. In contrast to that, we use goal functions as the strategy's actions, which have to be realized by a separate skill-learning layer. This leads to a perfect separation of concerns: the task of the strategy layer is to find sequences of actions and treats actions as mere symbols. The skill layer by means of data driven skill functions then grounds these symbols.

Another aspect is the supported number of goals. Take for example a system, which has to fulfill a specific task while paying attention to its diminishing resources. While, on the one hand, accomplishing the task, the resources might get exhausted. If it, on the other hand, always stays near the fuel station, the task will not be accomplished. Approaches like AMPS, which do not support multiple goals by multiple separate strategies, have to incorporate all different goal aspects in one reward function. This leads to a combinatorial explosion in the state space and implicates a much slower learning convergence.

As already described, we use abstract motivations, which the designer has to specify. These motivations may also contain competing goals. The major advantage of our approach is that the robot can learn one separate strategy for each motivation. Depending on the strength of each motivation, it has now a means to choose the right strategy for the actual perception and motivation state.

### C. Discussion

All these approaches have the following underlying restricting assumptions. First, they assume that optimal actions are either possible to be predefined or effectively learnable within the reinforcement learning framework. That means that prior to using these approaches a careful analysis of all occurring events in the environment has to be carried out by the designer. Except for AMPS, they are all based on Markov Decision Processes (MDP). Time varying actions, which are the norm in realistic scenarios, however require a semi-Markov Decision Process (SMDP), which complicates the search in continuous action spaces. Arguing that models are difficult to approximate at runtime the model-free approaches do not learn a model on which the policy is approximated but only the value function. Furthermore, they always solve only one goal and it is not intuitively clear, how multiple possibly contradicting goals could be integrated using the same state and action space for all goals. The biggest problem of all, however, is that these approaches are solely aimed at learning from scratch. It is not clear how those could be combined with imitation or coordination – aspects, which are vital to application multi-robot scenarios. The ESLAS architecture, which will be described in the following, was designed with these aspects in mind.

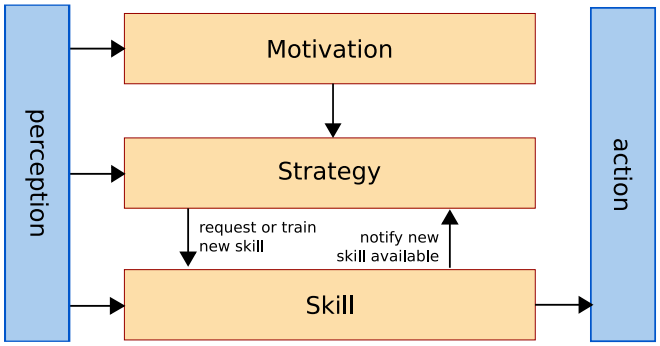


Fig. 1. The architecture of ESLAS consists of three layers. Every layer can read the perception and send its output to the module below itself

## III. The ESLAS architecture

Vital to the multi-robot imitation approach is the *Evolving Societies of Learning Autonomous Systems* (ESLAS) architecture that supports it by means of its layered recognition approach [9]. Thereby, we extend our previous layered architecture [1] to be usable for imitation in multi-robot scenarios.

The ESLAS architecture is based on three layers of abstraction as shown in Fig. 1. At the top level, a motivation layer provides a motivation function for the learning algorithm being the overall goal of the robot. This function determines which goal is the most profitable one to reach at each moment. With different motivations, the learning algorithm is able to handle changes in the environment without the need of relearning everything. At the medium layer is the Reinforcement Learning algorithm, which incorporates the method from AMPS of state space revising in parallel to SMDP policy calculation. It receives input from the interface and decides which skill is executed. A skill is described by a goal function and handled in the lowest layer. Skills can be simple, like driving forward, but also quite complex, depending on this function. Using this function, a skill is also capable of recognizing whether a skill similar to itself has been executed in the observations.

### A. Motivation layer

For the evaluation of the robot's overall state, we use biologically inspired evaluation methods similar to emotions. With that, we specify all high-level goals in the form of a motivation system (Fig. 2):

$$\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)^T \ , \quad \mu_i \in \mathbb{R}^+ \ . \tag{1}$$

Each motivation $\mu_i$ corresponds to one high-level goal, which is considered accomplished or satisfied if $\mu_i < \mu_i^\theta$, with $\mu_i^\theta$ defining the threshold of the *well-being region* (Fig. 3). By specifying $\mu_i : \mathbb{S} \rightarrow \mathbb{R}$ as a mapping from the strategy's state space to the degree of accomplishment of goal $i$ and $\mu_i^\theta$ as the satisfaction-threshold of that goal the designer is able to intuitively define the robot's overall goal, which it accomplishes by minimizing each motivation's value. When it is adapting its strategy or skill set, it does so with only this urge in mind.

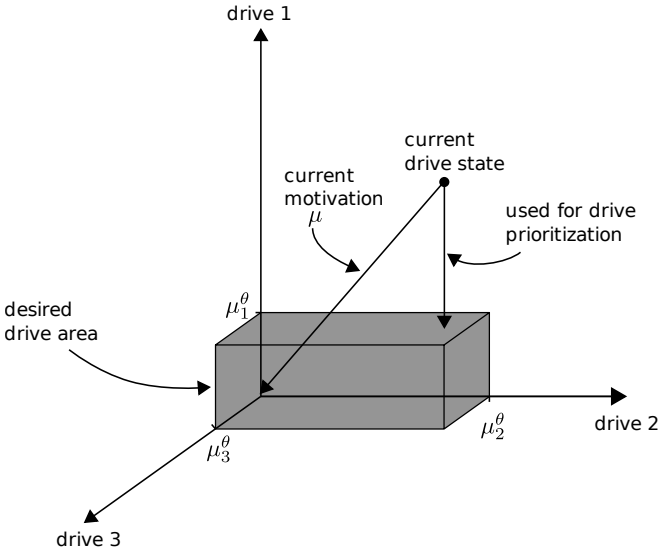


Fig. 2. The motivation system: each drive measures the status of accomplishing one sub-goal with zero being fully accomplished. The current motivation is the vector to the point of origin

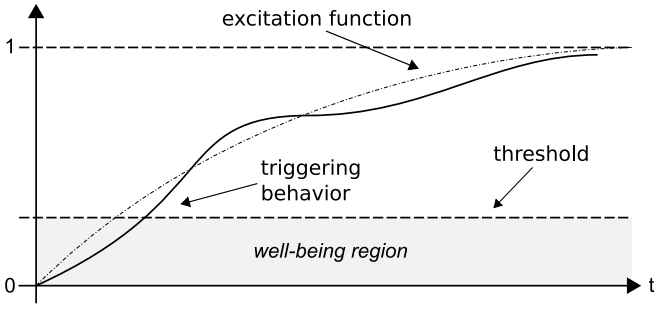


Fig. 3. An example of specifying a sub-goal by means of the motivation system's drive. The excitation function describes the force the current drive state is subject to. By specifying it dependent on the perception and on the internal state of the robot the user is "programming" the final behavior

If the vector of the current drive state to the point of origin is interpreted as the current motivation, it serves two functions in the ESLAS framework: on the one hand $-\dot{\boldsymbol{\mu}}$ is used as a reward for the strategy layer, which will be described in the next section. On the other hand, it supports imitation in multi-robot scenarios: the motivation value in this motivation layer can be used to express the robot's overall well-being to the other robots and guides them when they are observing each other to imitate only obviously beneficial behavior.

### B. Strategy layer

In order to satisfy the motivation layer the robot has to learn a strategy that is able to keep $\boldsymbol{\mu} < \boldsymbol{\mu}^{\boldsymbol{\theta}}$, given only the experience stream

$$\ldots, (o, a, d, \boldsymbol{\mu}, f)_{t-1}, (o, a, d, \boldsymbol{\mu}, f)_t, \ldots \tag{2}$$

where $o_t$ is the raw observed state, $a_t$ the executed action triggered in the last time step, $d_t$ the duration of that action, $f_t$ signals whether the action has failed, which will be

described later on. To keep this learning program tractable the strategy layer is not trying to learn one strategy for the whole motivation system. Instead, it is generating one strategy for each motivation. The system then selects the strategy to follow dependent on the dynamic drive prioritization $\max(\mathbf{0}, \boldsymbol{\mu} - \boldsymbol{\mu}^{\theta})$. In the current approach it chooses the drive with the least satisfied motivation.[1]

In the following, we will restrict the description to one strategy. It will have to generalize the actual state observations into abstract regions on which it then uses Reinforcement Learning to find a sufficiently good strategy, as operating on the raw state space would be unfeasible. Thereby, it can be used with any form of abstraction method. In this work, we use nearest neighbor [10]. The environmental model is updated during runtime as new experience is made by the robot and thus subject to change. A model-based Reinforcement Learning with prioritized sweeping [11] is used to derive an optimal policy by means of semi-Markov decision processes (SMDP) [12], [13]. Model-free Reinforcement Learning methods like Q-learning [14] are not practicable in this case, because all experience gets lost each time the underlying model changes.

We will now present the three main components of the strategy layer and their interaction (Fig. 4): the processed, filtered and purified perception is stored as an interaction sequence, which we will call *experience*. It is modified by several heuristics to build a *model* upon which the *policy* is generated. We will start explaining how the strategy is learned, if the model has already been built so that it reflects the experiences and abstracts the raw state observations $o$ to abstract state regions $s \in \mathbb{S}$.

*1) Policy:* A policy is a mapping

$$\pi : \mathbb{S} \rightarrow \mathbb{A} \tag{3}$$

that assigns each state $s \in \mathbb{S}$ an action $a \in \mathbb{A}$. Let $V^{\pi} : \mathbb{S} \rightarrow \mathbb{R}$ be a value function that estimates "how beneficial" it is for a robot to be in a given state. A policy $\pi$ is called optimal, if $V^{\pi}(s) \geq V^{\pi'}(s)\ \forall\ s \in \mathbb{S}$. Thereby, it maximizes the expected long-term discounted sum of rewards [13] and is denoted by $\pi^*$. The reward is discounted, as it is wise to give a smaller weight to reward that is further away in the future.

As in real-world scenarios the duration of actions are variable, the discounting is done continuously by $\beta \in (0, \infty)$: a reward $r$ received after time $t$ thus leads to a net reward of $e^{-\beta t}$. If $\beta = \infty$ the robot is said to be myopic, as the future reward is discounted by $e^{-\infty t} \approx 0$ and the robot thus is concentrating only on the immediate reward. With $\beta$ approaching zero the robot is paying more and more attention to reward that is farther in the future.

The reward in our work is composed of two reward elements. The lump sum reward $r$ specifies the one time reward for transferring the robot from the abstract state $s$ with action $a$ to the abstract state $s'$. The reward rate $\rho$ is given continuously for staying in state $s$ while executing action $a$ until the robot arrives at state $s$. This is necessary to provide the most general form of goal specification via the motivation system. Both components can be extracted from the motivation by means of

$$r, \rho = \begin{cases} (-\dot{\mu}_i, 0) & \text{if } |\dot{\mu}_i| > \rho_{\theta} \\ (0, -\dot{\mu}_i) & \text{otherwise} \end{cases} . \tag{4}$$

That means that the reward is interpreted as lump-sum reward, if it exceeds the reward rate threshold $\rho_{\theta}$, otherwise it is received as reward rate.

In the following, we will stick to the notation of Kochenderfer regarding the learning of strategies on abstract state spaces with SMDPs. With $P_t(t \mid s, a, s')$ being the probability that it takes at most $t$ time to move from $s$ to $s'$ by means of executing action $a$ the *discounted value of the unit lump-sum reward* is calculated as[2]

$$\gamma(s, a, s') = \int_0^{\infty} e^{-\beta t} dP_t(t \mid s, a, s') . \tag{5}$$

The *average cumulative discounted sum of the reward* received continuously while executing action $a$ in state $s$ until arriving at state $s'$ is calculated as

$$\lambda(s, a, s') = \int_0^{\infty} \int_0^{t'} e^{-\beta t'} dt' dP_t(t \mid s, a, s') . \tag{6}$$

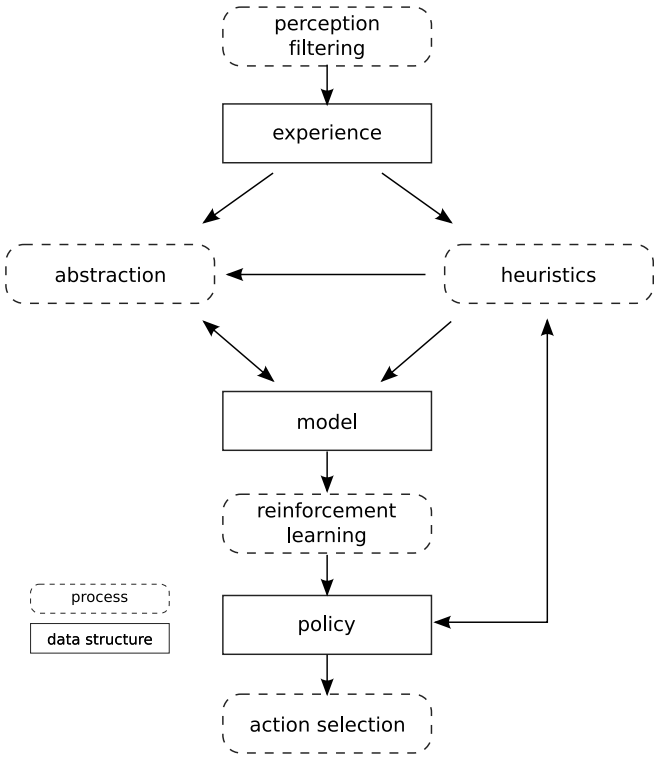


Fig. 4. Processes involved in the strategy layer

[1] In addition, we are investigating methods, which try to detect situations in which compromises are made by choosing a possibly suboptimal action for one motivation, if others can be pleased with that action as well.

[2] Not to be confused with the discount factor $\gamma$ in MDP problems.

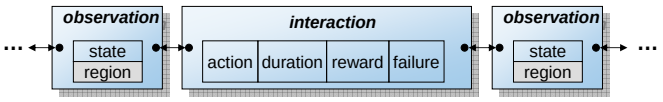


Fig. 5. The interaction sequence forms the experience flow

The expected discounted reward when started in state

$$V_\pi(s) \equiv E\Bigg\{ \sum_{k=1}^{\infty} \left[ e^{-\beta t_{k+1}} r_k + \int_{t_k}^{t_{k+1}} e^{-\beta t} \rho_k dt \right] \Big| \quad (7)$$

$$s_1 = s, a_k = \pi(s_k) \Bigg\}. \quad (8)$$

To find the optimal value function $V^*(s)$, the robot is updating recurrently the value function each time a new event occurs using

$$V(s) \leftarrow \max_{a \in \mathbb{A}} \left[ R(s,a) + \sum_{s' \in \mathbb{S}} P(s'|s,a)\gamma(s,a,s')V(s') \right]. \quad (9)$$

Eventually, $V(s)$ will then converge to the true value function $V^*(s)$ [8]. The optimal policy can then be computed similarly:

$$\pi^*(s) = \arg\max_{a \in \mathbb{A}} R(s,a) + \sum_{s' \in \mathbb{S}} P(s'|s,a)\gamma(s,a,s')V^*(s') \quad (10)$$

The only thing left to do is to estimate $R(s,a)$ ("$(s,a,s')$" omitted):

$$R(s,a) = \sum_{s' \in \mathbb{S}} P(s' \mid s,a) \left(\gamma r + \lambda \rho\right) \quad (11)$$

Kochenderfer showed that this can be done with non-parametric estimation [8]. For that it is first necessary to estimate $\gamma(s,a,s')$, as $\lambda(s,a,s')$ simplifies to $\lambda(s,a,s') = (1 - \gamma(s,a,s'))/\beta$. If $n(s_k,a_k,s_{k+1})$ is the number of $(s_k,a_k,s_{k+1})$ transitions, then $\gamma$ is estimated after the $k^{th}$ transition as follows:

$$\hat{\gamma}(s_k,a_k,s_{k+1}) \leftarrow \hat{\gamma}(s_k,a_k,s_{k+1}) + \frac{e^{-\beta t_k} - \hat{\gamma}(s_k,a_k,s_{k+1})}{n(s_k,a_k,s_{k+1})} \quad (12)$$

If $\sigma_r(s,a,s')$ is the accumulated lump-sum reward and $\sigma_\rho(s,a,s')$ the sum of the reward rates received when going from $s$ to $s'$ with action $a$, then the estimated expected reward for executing $a$ in $s$ can be calculated as

$$\hat{R}(s,a) = \frac{1}{n(s,a)} \sum_{s' \in \mathbb{S}} \left(\hat{\gamma}(\sigma_r - \sigma_\rho/\beta) + \sigma_\rho/\beta\right) \quad (13)$$

*2) Experience:* To arrive at a tractable number of meaningful states, the raw states have to be abstracted first. The ESLAS approach works on interactions that describe the action in and the reaction of the environment (Fig. 5). An interaction encodes the following data:

- *action:* This is the output that was delivered to the action tower in the last step.
- *duration:* The strategy is not informed every time new information is available. Instead, it is triggered using intelligent heuristics, which will be described later on.
- *reward:* The return of the last action in the form of the motivation vector.
- *failure:* This is retrieved from the skill layer and denotes whether the skill executed in the last step estimates the outcome as success or failure. It will be described in more detail in Sec. III-C.

An interaction is always connected to its starting and ending states (Fig. 5) provided by the perception. All the experience is saved in an experience list, which consists of *interactions*. We define an interaction to describe the important data of one time frame

$$I_{t_1}^{t_2} = (o_{t_1}, a_{t_1}, d_{t_1}, r_{t_1}, f_{t_1}, o_{t_2}), \quad (14)$$

where $o_{t_1}$ and $o_{t_2}$ are the raw state observations at the beginning and end of a time frame (*state* in Fig. 5). $a_{t_1}$ is the executed action, $r_{t_1}$ the reward vector received by the motivation layer and $d_{t_1} = t_2 - t_1$ the duration. Finally, $f_{t_1}$ denotes a failure of the last step. This can be e.g. the skill layer signalling that the previously executed skill has not performed as expected, because the robot is trying to drive against a wall. For realistic applications, it must be taken care that the robot is not spammed with uninteresting information. A new interaction is generated if one of the following heuristics holds:

- The perception signals a sufficiently different state by some distance metric: $d(o_{t_1}, o_{t_2}) > \theta_o$.
- The motivation layer has signaled a sufficiently interesting motivation change: $r_{t_1} = |m_{t_2} - m_{t_1}| > \theta_r$
- A certain amount of time has passed: $t_2 - t_1 > \theta_t$

*3) Model:* At the beginning, all states belong to only one region, as the robot has no reason to believe otherwise. While interacting with the environment the model is modified by several heuristics, which are invoked recurrently to maintain a mapping of observations in the perception space $\mathbb{R}^d$ (*state* in Fig. 5) to states in the abstracted region space $\mathbb{S}$ (*region* in Fig. 5). $d$ is the number of dimensions of the perception space[3]. The heuristics split or merge regions so that the model and underlying statistics reflect the world experience. The following heuristics are found to be necessary.

*a) Transition heuristic:* As mentioned above, the continuous state space is split into regions so that for each raw state belonging to the same region executing the same action "feels" similar to the robot. That requires that $Q(s,a,s')$ as the value for transitioning from $s$ to $s'$ with the greedy action $a = \pi(s)$ can be estimated with a sufficient confidence. This is calculated using interaction sequences starting in $s$ and arriving in $s'$ while only executing the greedy action $a$:

$$Q(s,a,s') = \gamma(s,a,s')(r(s,a,s') + V(s')) \quad (15)$$

$$+\lambda(s,a,s')\rho(s,a,s') \quad (16)$$

Let $succ_a(s) = \{s' \mid P(s' \mid s,a) > 0\}$. If raw states are mistakenly grouped into the same abstract region the variance of the $Q(s,a,s')$ values calculated for all the greedy traces

[3]In the experiments nearest neighbor is used. Practically any abstraction mechanism can be used that supports add/remove/query at runtime.

belonging to the same region will increase. A high variance indicates that splitting that region will likely lead to better transition estimates in the split regions:

$$Var\left(\{\, Q(s,a,s') \mid s' \in succ_a(s) \,\}\right) > \theta_{TV} \qquad (17)$$

This is done by clustering the traces so that traces with similar $Q(s,a,s')$ grouped together. For each cluster, one region is created.

The challenge lies in determining $\theta_{TV}$. AMPS requires the designer to analyze the scenario and empirically determine that value beforehand. This is apparently no possibility for groups of robots, which have to learn the proper behavior autonomously themselves. As the distribution for $Q$ usually cannot be foreseen it happens that $\theta_{TV}$ is either to low, which results in too fine state abstraction and slows down the learning speed, or too high which leaves too much aliasing in the strategy. The competing forces for determining $\theta_{TV}$ are as follows:

1) The more often the robot is experiencing aliasing and the higher the variance of the resulting regions' values is, the higher the inclination to split should be.
2) The lower the variance is compared to the maximum region value the lower the inclination to split should be.

The first point is solved by using $QVar$ which weights the deviation from the mean by the region's transition probability. The second is handled by normalizing both mean and the $Q$-values to the maximal region value $V_{max} = \max(\{V(s) \mid s \in \mathbb{S}\})$. With the following definition for $QVar$ the threshold $\theta_{TV}$ can be set to a fixed value without having to bother about the future development of the region values:

$$QVar\left(Q(s,a,s')\right) \equiv \sum_{s' \in succ_a(s)} \frac{P(s' \mid s,a)}{V_{max}^2 \cdot |succ_a(s)|} \cdot \left(\overline{Q(s,a,s')} - Q(s,a,s')\right)^2 \qquad (18)$$

The inclination to split is thus adapting with the changing value function at run-time.

*b) Experience heuristic:* This heuristic limits the memory horizon of the robot to $\theta_M$ interactions. It removes interactions that are too far in the past in order to keep the robot's model and policy more aligned to the recent experience of the robot. Basically, it removes those old interactions from its memory and adds the new experience to it. Thus, it is modifying the experience of at most two regions, which might cause an update of the model and of the policy.

*c) Failure heuristic:* A failure rate is associated with each region. It describes the ratio of failure signals when the greedy action of the corresponding region has been executed to the number of success signals. These signals are emitted by the strategy and skill layer which will be described later on. They are encoded as $f_t$ in the interactions (Eq. (14)). Failure signals are scenario specific and can be emitted if e.g. the robot bumps into a wall or if it has not encountered something interesting for a longer period of time. The failure heuristic splits a region if its greedy action's failure rate is not homogeneous enough:

$$\theta_f < f < 1 - \theta_f, \quad (0 < \theta_f < 1/2) \qquad (19)$$

The lower the user defined threshold $\theta_f$ is, the more eager the failure heuristic is trying to split a region. This forces the state abstraction to arrive at regions that have failure rates with which a more deterministic strategy can be computed. For both resulting new regions individual greedy actions can then be determined by the reinforcement learning algorithm.

*d) Reward heuristic:* Especially in the beginning of the robot's lifetime, when there is not yet enough information for the transition and simplification heuristic to adapt the state space based on sufficient statistical data, the reward heuristic is of importance. It allows a region $s \in \mathbb{S}$ to be split if the reward rate variance is too high. This indicates that the action performed in that region gives a too diverse feedback. A split of that region will then lead to multiple regions, which are more consistent with regard to the expected reward rates. This also is vital in cases where the failure signal is too seldom, as it provides the only other possibility to initially split a region.

In particular, the reward heuristic is looking in the reward rate stream for a clear switch from low to high variance areas, where both areas are of sufficient length. Only such a switch in variance indicates clearly that a split is advisable. Therefore the reward heuristic considers the reward rates of the last $n$ interactions made in the current region. The lump sum rewards in that time frame are not considered, as they will show non-zero values only in rare occasions. Let $\rho_{t_1}^{t_2} = (\rho_{t_1}, \ldots, \rho_{t_2})$ and $t$ be the time at which the split is considered. The reward heuristic is searching for an index $k$ that splits $\rho_{t-n}^{t}$ into the two sequences $\rho_{t-n}^{t-k-1}$ and $\rho_{t-k}^{t}$, such that the following condition holds:

$$\begin{gathered}\left(Var(\rho_{t-n}^{t-k-1}) \approx 0 \ \wedge \ Var(\rho_{t-k}^{t}) > \theta_{RV} \ \wedge \ |\rho_{t-n}^{t-k-1}| > \theta_l\right) \\ \bigvee \\ \left(Var(\rho_{t-n}^{t-k-1}) > \theta_{RV} \ \wedge \ Var(\rho_{t-k}^{t}) \approx 0 \ \wedge \ |\rho_{t-k}^{t}| > \theta_l\right)\end{gathered} \qquad (20)$$

The minimum variance threshold $\theta_{RV}$ is dependent on the motivation system design. Recall from Sec. III-B.1 that the reward received by the motivation system is interpreted as a reward rate, if $|\dot{\mu}_i| \leq \rho_\theta$. With $\theta_{RV} = k \cdot \rho_\theta$, $(0 < k < 1)$, a switch is easily detected by the reward heuristic. The minimum low variance sequence length $\theta_l$ ensures that the reward heuristics does not find trivial splits. Naturally it is set to be a fraction of the considered time horizon $n$.

*e) Simplification heuristic:* As splitting might lead to overly complex models a means is needed that again merges regions once the robot has gathered new experience that suggests a simpler model. This is the task of the simplification heuristic, which analyzes sequences of regions connected by greedy actions. Similar to AMPS we consider here chain and sibling merges. Let $a$ behave nearly deterministically in $s$, then $succ(s,a)$ denotes the region the execution of $a$ leads

to:

$$succ(s,a) \equiv \begin{cases} s' & \text{if } P(s'|s,a) \approx 1 \\ \text{None} & \text{otherwise} \end{cases} \quad (21)$$

A chain merge of two regions $s'$ and $s'$ is performed if

$$succ(s', \pi(s')) = s'' \wedge succ(s'', \pi(s'')) = s \wedge \pi(s') = \pi(s'') \; . \quad (22)$$

In this case the region $s''$ is superficial and can thus be merged with $s'$ into the new region $s''' = s' \cup s''$, with $succ(s'''', \pi(s''')) = s$ and $\pi(s''') = \pi(s`) = \pi(s'')$. All other regions that resulted into either $s'$ or $s''$ are updated accordingly.

In the same vein a sibling merge is triggered if

$$succ(s', \pi(s')) = s \wedge succ(s'', \pi(s'')) = s \wedge \pi(s') = \pi(s'') \; . \quad (23)$$

In this case $s'$ and $s''$ have similar expectations about the future region if the same action is executed.

So far we have assumed that $\mathbb{A}$ is always provided beforehand and that the strategy simply has to choose the right action at each state. For real-world scenarios it would be advantageous if also $\mathbb{A}$ could be learned at run-time. AMPS does this by applying to $\mathbb{A}$ the same abstraction heuristics that helped to organize the state space $\mathbb{S}$. The actions learned in this way, however, are limited to simple domains, where the real-world dynamics can be presented by simple hypotheses. In the next section the *Automatic Modular Action Framework* (AMAF) is presented, which is able to learn reactive actions that are robust to noise and can handle complex dynamics.

### C. Skill layer

The skill layer provides a generalized learning method for learning reactive low-level skills. It offers to the strategic layer two working modalities, one for training new skills and one for executing one of the learnt skills. As long as no skills are available, the skill layer explores the space of the low-level actions by composing with random values the output vector that will be sent to the actuators.

Each learnt skill allows to control the perceived properties of the environment by continuously associating an error value to the input data. The learnt skills are communicated to the strategic layer through the identifier that will be used to handle the skill and the definition of the skill.

When the execution of a skill is requested, the skill layer reacts to the received inputs with low-level actions (output vectors) that minimize the error described in the definition of the skill.

For our strategy-learning algorithm, we assume that all skills have finished building hypotheses and are ready for execution.

There are two major benefits with the skill layer over using atomic actions. First, it gives the possibility to automatically create quite complex actions that evolve and adapt to changes. Second, a skill can recognize itself out of a trace of observations by monitoring the value of the error. If this decreases, it is reasonable to deduce that the action was in execution. This is how we solve the correspondence problem between observed execution of foreign behavior and own capabilities in our scenario [15].

The skill layer has been implemented using AMAF (Automatic Modular Action Framework), a framework for the automatic creation of abstract actions. The generated abstract actions will be used as skills by the strategy. What follows is an overview of the framework.

## IV. AMAF Specification

### A. Working modalities

AMAF can work in two modalities: the passive (or execution) and the active (or training) modality:

- *Execution:* during the execution modality, AMAF is passive because the robot decides which action to execute.
- *Training:* during the training modality, AMAF is active because it decides which action to execute in order to experiment new actions.

This distinction has not to be confused with the one between learning and acting. Learning and acting are parallel processes indeed: AMAF supports the learning phase during both the execution and training modalities. During both modalities, AMAF has to react to the input data with an output vector, so the acting is always enabled.

### B. Environment structure

AMAF works with structured data in order to have more abstract and powerful actions and to make a faster and more general learning possible. The robot's perception has to be structured in *objects* and *properties* in order to be used by AMAF. A property is an attribute of an object recognized in the environment. It is composed by an ID and a value, expressed by a real number. An object is an element of the environment characterized by an ID and a set of properties.

The input data has to be structured as a list of tuples $\langle id_o, id_p, v \rangle$ where $id_o$ is the identifier of an object $o$ in the environment, $id_p$ is the identifier of a property $p$ of $o$ describing one perceived attribute of that object, and $v \in \mathbb{R}$ denotes its value.

### C. Output representation

AMAF generates abstract actions starting from the actions of the lowest level of abstraction. Usually in robotic applications, the low-level action is the set of intensities of the electrical signals sent to the actuators or, in the case of servomotors, the sent value. A general way to interpret the information sent to an actuator is the effort that actuator will make. AMAF represents a low-level motor action by the output vector $\boldsymbol{M} = (m_1, \ldots, m_n)$, with $m_i \in \mathbb{R}$, $-100 \leq m_i \leq 100$, indicating the effort that a certain actuator will make, and $n$ being the number of actuators the robot controls.

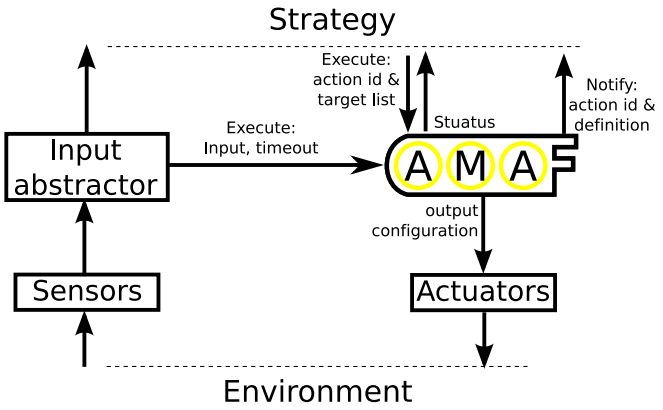


Fig. 6. AMAF interaction with the other components. The row input data is supposed to be elaborated by a specific component called "input abstractor"

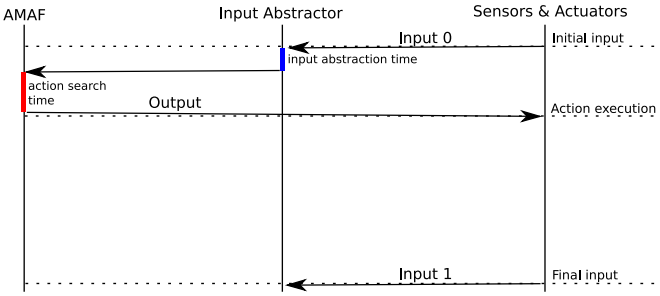


Fig. 7. Example of low level interaction

### D. Interfaces

The strategy communicates with AMAF by sending the action to be executed. There is just one pre-built action with ID *train* that switches AMAF to the active modality. All the other actions are learned by AMAF during its execution. The robot executes a learned action by sending to AMAF the action ID and the list of the targets of the action. These are expressed by the IDs of the objects that the action is applied to. During this phase, the robot can monitor the status of the current action. When the action is set, the status is *starting*. After the first low-level interaction with the environment the status can be *failure* if it is not possible to execute the action, *execution* if the action is executed at low level but the success condition is not verified or *success* if the success condition has already been reached. During both active and passive phases, AMAF can decide that an action executed during the training phase is *mature*. This indicates that it is ready to be immediately used by the robot. In this case, the action is notified to the robot by sending the action ID and the action definition.

At the interface to the environment, AMAF receives the abstract input data and returns the output configuration that realizes the abstract action. It has to compute the solution in a certain time interval that is dependent on the frequency of interaction with the environment. In order to react meaningful in time, AMAF receives in addition to the abstract input the timeout for searching the best low-level action. After the timeout, AMAF has to return the best output configuration found by then, even if it is not globally the best for the last received input.

Fig. 7 gives an example of a low-level interaction. The sensors send the raw input data to an external component, called "Input Abstractor". This processes the raw input and computes information usable for AMAF in a certain time interval. A timeout determines how much time AMAF has for calculating the output configuration. Finally, the output configuration is sent to the actuators. The interaction cycle starts again when a new input is perceived from the sensors and sent to the Input Abstractor.

For each interaction with the environment, AMAF assumes that the next input is the effect of the returned output on the actual input. This can be a good approximation only when the time interval between the initial input perception by the sensors and the execution of the output configuration by the actuators is not relevant in comparison to the time interval between the two input perceptions. We can usually consider unimportant the latencies of the sensors, of the actuators and of the communication between the components. It is sufficient to take into account just the computational time of the Input Abstractor $\Delta T_I$ and of AMAF $\Delta T_M$, and the time interval between two inputs $\Delta T_i$. More formally this means that $\Delta T_I + \Delta T_M \ll \Delta T_i$.

### E. Actions

The aim of AMAF is to automatically generate abstract actions that allow the robot to move easily in the state space and accomplish its tasks. Each robot, however, can have a different state space representation. The way, in which the input variables are interpreted and combined together, varies significantly from the different implementations of the robot. I.e., AMAF cannot directly control the robot state through its actions. The AMAF solution is to control the input variables. The robot state space is directly built on their values so actions that control the input variables, indirectly allow any kind of robot to move freely in its state space.

Each perceived property of an object is controlled by a "basic action". In addition, AMAF is able of generating "complex actions" that coordinate the execution of different basic actions. The actions generated by AMAF are *multi-target* because it is possible to specify the list of identifiers of the objects that are target of the action. In this way, the complex action can control the value of the properties of different objects at the same time. We will now give a detailed definition of basic and complex actions.

*1) Basic Actions:* The elementary blocks of the actions supplied by AMAF are the "basic actions". A basic action is defined by a tuple $\langle c, p \rangle$ where $p$ is a property of the object specified as target and $c$ indicates a control. A *control* is a function $f_c : \mathbb{R}^2 \rightarrow \mathbb{R}^+$ that associates an error to each tuple $\langle iv_p, av_p \rangle$ where $iv_p$ is the value of $p$ when the action was started and $av_p$ is the current value of $p$. The function $f_e : \mathbb{R} \rightarrow \mathbb{R}^+$ obtained by fixing the value $iv_p$ is called *error function* of the basic action. During the execution of the basic action, AMAF tries to decrease as

much as possible the value of the error so different controls determine different behaviors. E. g., if $f_e$ is proportional to the value of the property, the basic action decreases its value. On the opposite if $f_e$ is inversely proportional to the value of the property, the basic action increases its value. It is even possible to have actions that control the variation of the value of the property. E. g., in order to specify an action that increases $iv_p$ of an interval $\Delta p$ it is sufficient to use a control $f_c(iv_p, av_p) = |iv_p + \Delta p - av_p|$.

*2) Complex Actions:* Basic actions allow the robot to move in the state space but sometimes there can be performance requirements that cannot be satisfied by simply executing basic actions in sequence. It can be for instance necessary to execute two basic actions at the same time or to start executing one action when another one is going to finish. Imagine a task that requires to get close to an object and to shoot it. In this case it is necessary at first to reduce the distance to the object then it is necessary to center the object and finally to shoot. The result can not be efficient if the actions are executed in sequence, but if the object is reached while centering it, the chances of success increase. In AMAF it is possible to coordinate different basic actions through the "complex actions".

The number of targets and the ordered sequence of steps define a complex action. Each step is defined by a tuple $\langle l_R, t_R, s_R, l_A \rangle$ where $l_R$ is a list of references that determine the *reference error*, $t_R \in \mathbb{R}^+$ is the *success condition* threshold, $s_R \in \mathbb{R}^+$ is the *precondition* threshold and $l_A$ is the list of the weighted basic actions. The elements of the list $l_R$ are tuples $\langle c, p, t, w \rangle$ where $c$ is a control, $t \in \mathbb{N}^+$ indicates the index of the object $o$ of the target list, $p$ is a property of $o$ and $w \in \mathbb{R}^+$ is the weight of the reference. The reference error is computed by linear combination of the errors obtained by applying the controls on the input data. The weights are used as coefficients of the linear combination. The value of reference error $V_r$ determines the value $V_{pl}$ of the progress level:

$$V_{pl} = \begin{cases} 0 & \text{if } V_r > s_R \\ \frac{s_R - V_r}{s_R - t_R} & \text{if } t_R < V_r < s_R \\ 1 & \text{if } V_r < t_R \end{cases} \quad (24)$$

When the value of the progress level is one, the success condition of the step is reached and the execution of the step can be considered completed.

The elements of $l_A$ are weighted basic actions $a$ defined by the tuple $\langle c_a, p_a, t_a, cf_a \rangle$ where $c_a$ is a control, $t_a \in \mathbb{N}^+$ indicates the index of the object $o$ of the target list, $p_a$ is a property of $o$ and $cf_a : [0,1] \rightarrow [0,1]$ is the coefficient function that determines the value of the coefficient for each value of the progress level. The coefficient function $cf_a(V_{pl}) = start + f_s(V_{pl}) \cdot scale$ is defined by the tuple $\langle id_s, scale, start \rangle$ where $id_s$ is the identifier of the shape function $f_s : [0,1] \rightarrow \mathbb{R}$, $scale \in \mathbb{R}$ is the scaling factor and $start \in \mathbb{R}$ is the starting value of the coefficient. The value of the coefficient $c_a$ of the basic action $a$ is determined by

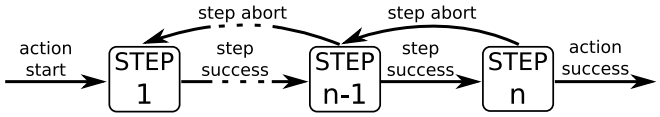


Fig. 8. A complex action is composed by a sequence of steps

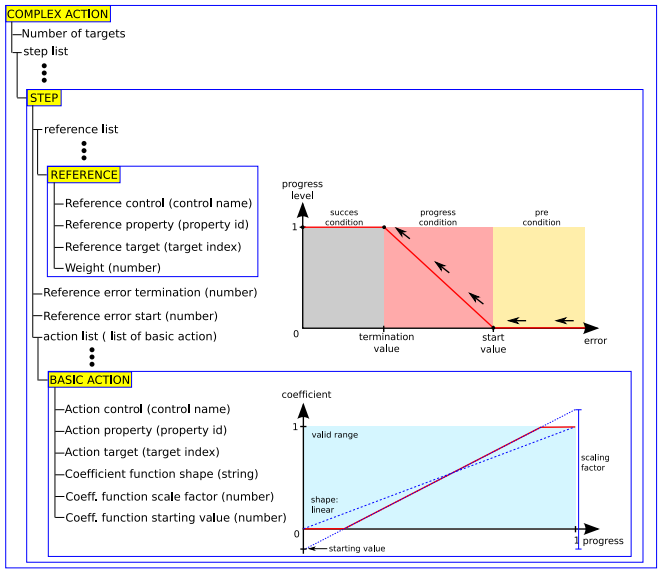


Fig. 9. The structure of the definition of a complex action

coefficient function and the progress level:

$$c_a(V_{pl}) = \begin{cases} 0 & \text{if } cf_a(V_{pl}) < 0 \\ cf_a(V_{pl}) & \text{if } 0 \leq cf_a(V_{pl}) \leq 1 \\ 1 & \text{if } cf_a(V_{pl}) > 1 \end{cases} \quad (25)$$

Calling $A$ the set of basic actions $a$ executed during the step, the error function of the step is

$$f_s = \sum_{a \in A} c_a(V_{pl}) e_a(p_a) \ , \quad (26)$$

where $e_a$ is the error function of $a$.

The action starts with the execution of the first step. When the success condition of the step is reached, the second step is executed and so on until the last step. The success condition of the complex action is reached when the last step is in its success condition as shown in Fig. 8. The structure of the definition of a complex action is shown in Fig. 9.

### F. Framework structure

The modules of AMAF are divided into two groups: the learning modules and the performing ones. The former have to learn from the interaction with the environment and with the robot. Each learning module represents its knowledge through specific elements called *knowledge units* and gives a score to each of them. The performing modules for the execution of both passive and active modalities use these elements.

The performing modules directly communicate through buffers while the information flow of the learning process is based on the CENTRAL LOG SYSTEM . This is a component

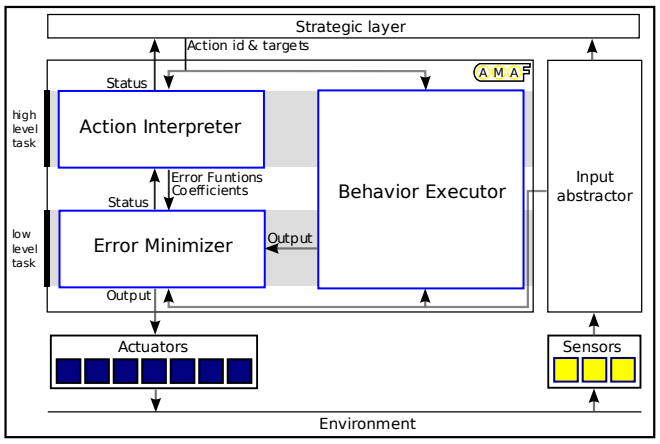


Fig. 10. Information flow of the performing modules of AMAF

structured into logs that stores all the information communicated by the performing modules and all the knowledge units with their scores. All the modules can read from all the logs of the CENTRAL LOG SYSTEM . This mechanism makes the communication inside of the framework flexible and permits the lack of synchronization between the performing modules and the learning ones.

AMAF works at two levels of abstraction (Fig. 10). At the high abstraction level, an action is expressed by an error function. At the low abstraction level, the action is expressed by the output vector that will be sent to the actuators.

These two divisions determine four subtasks each solved by a specific module. Finally there are two extra modules that are not necessary for the functioning of the framework but allow a significant improvement of the timing performance by modeling the behavior of the action through a direct mapping of the low-level action to the input data. For this motivation, these two modules work at the same time at both high and low abstraction levels.

- ACTION INTERPRETER learns at the high abstraction level. It has to transform the abstract action received from the strategy layer into an error function. If the action is the special action *train*, the ACTION INTERPRETER has even to decide, which action to execute between the ones defined by the ACTION MANAGER . During the execution modality, it has to determine the status of the abstract action.
- ACTION MANAGER learns at high abstraction level. It creates new actions and improves the learned ones. The score of an action indicates how interesting it is to execute that action during the training modality. The new actions are initially not sent to the robot. That way they can be executed only during the training phase. An action will be notified to the strategy layer only when its performances during the training are considered sufficient.
- PREDICTION MODEL MANAGER learns at the low abstraction level. It creates and updates the prediction models that will be used by the ERROR MINIMIZER to predict the effect of a low-level action. A prediction model is defined by a tuple $\langle P, \boldsymbol{M}, p, f \rangle$, where $P$ is a subset of cardinality $m$ of the perceived properties, $\boldsymbol{M}$ is a subset of cardinality $n$ of output vector, and $p$ is the predicted property. $f$ is a function $f : \mathbb{R}^{m+n} \rightarrow \mathbb{R}$ that predicts the value that $p$ will assume at the next input perception by knowing the actual values of a $P$ and $\boldsymbol{M}$.
- ERROR MINIMIZER acts at the low abstraction level. It has to transform the error function in an output vector for each perceived input. The best output vector is the one that minimizes the error associated to the next received input. A time constraint can be set in order to compute the low-level action before the specified time interval.
- BEHAVIOR MODEL MANAGER learns to create and update the behavior models. A behaviour model is defined by a tuple $\langle P, \boldsymbol{M}, f \rangle$ where $P$ is a subset of cardinality $m$ of the perceived properties, $\boldsymbol{M}$ is a subset of cardinality $n$ of the output vector and $f$ is a function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ that computes the values of $\boldsymbol{M}$ by knowing the values of $P$. Each model has to reproduce the behaviour of a step of an abstract action. Its score indicates the capacity to reach the success condition of the step.
- BEHAVIOR EXECUTOR acts by using the behavior model associated to the actual step of the abstract action in execution to directly compute the output vector. No search in the low-level action space is needed so the computational time is drastically reduced. The computed value is sent to the ERROR MINIMIZER that can use it for finding the expected best output configuration.

### G. Configuring AMAF

AMAF can be tailored to the specific application through a few simple configuration possibilities:

- *Degrees of freedom*: the number of low-level actuators of the robot.
- *Controls*: AMAF requires the specification of the list of the controls used to generate the basic actions. Each of them can produce a basic action for each perceived property.
- *Modeling algorithms*: the creation of prediction models and of behavior models can be based on different modeling algorithms. AMAF requires at least one modeling algorithm. If more then one is specified, AMAF automatically chooses the modeling algorithm most appropriate for the current situation dependent on its prediction quality.

### H. Learning Flow

Each learning module continuously generates the definitions of the new knowledge units, updates the definitions of the previously existing knowledge units, and generates an updated ranking of the defined elements. A ranking is a list of tuples $\langle id_q, s_q \rangle$ where $id_q$ is the ID of an object $q$ and $s_q \in \mathbb{R}^+$ its score.

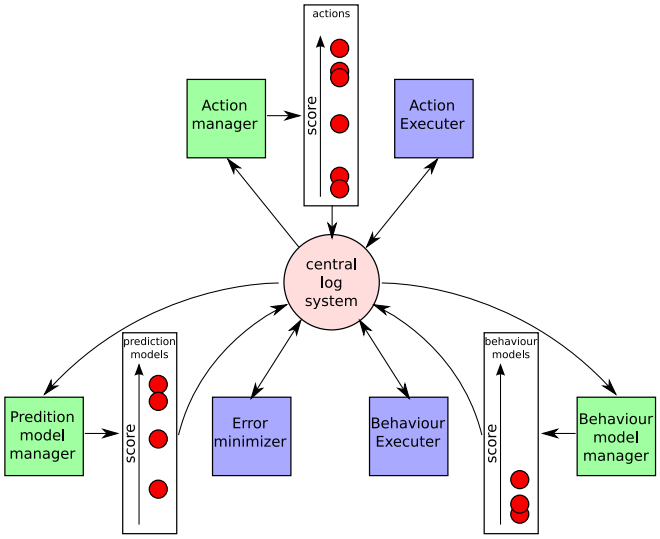


Fig. 11. Information flow of the learning process

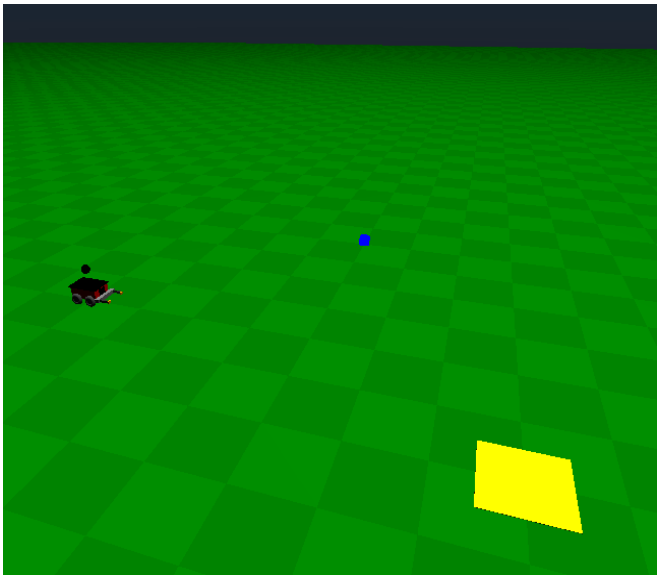

Fig. 12. Capture-The-Flag scenario. The robot has to learn to push the object to the yellow goal base. It has to learn by itself both the low-level actions and the strategy using them

The information flow of the leaning flow is shown in Fig. 11. The only constraint of the learning flow is that the definition of the knowledge units has to be compliant to the protocol specified by the framework.

For what concerns the information exchange, the CENTRAL LOG SYSTEM allows for a great variety in the implementation choices. The most trivial information exchange inside the learning flow is the one, in which the performing module uses the knowledge generated by the corresponding learning one.

The ERROR MINIMIZER has to continuously look inside the central log for new definitions of the prediction models and for updated rankings. Each time a perception arrives, the low level actions are evaluated by computing the expected future value of the properties controlled by the actual abstract action. The prediction models used to compute the expected values are the ones with higher score between the ones whose $P$ is a subset of the actually perceived properties.

During the active modality, the ACTION INTERPRETER has to decide which action to execute. It is possible to exploit the ranking generated by the ACTION MANAGER by choosing the action with higher score.

Exploiting the knowledge generated by a learning module could be useful not only by the associated performing module but also by a different performing module or even by another learning module. For example, the BEHAVIOR MODEL MANAGER could build the behavior models by computing the expected best low-level action for different input configurations and then modeling the results. The capability of predicting the effect of the low-level actions would be supplied by the prediction models generated by the PREDICTION MODEL MANAGER .

### I. Performing Flow

On the performing dimension, a fast and reliable information exchange is necessary in order to obtain a good reactivity. Therefore, it is not convenient to use a centralized system of the information exchange. AMAF manages the performing information flow by direct connections between the modules. The information flow is propagated from the higher abstraction level to the lower one. The work of each performing module can be monitored by reading its status.

## V. EVALUATION

In this section, we will present the use of ESLAS in a Capture-The-Flag scenario. In the PlayerStage/Gazebo [16] simulation (Fig. 12) the well-known Pioneer2DX robot is used. The dynamics are simulated using the *Open Dynamics Engine (ODE)* [17]. This scenario consists of a goal base to which pucks dispersed in the environment have to be transported. The robot has to find out which skills, autonomously learned by the skill layer (Sec. III-C), have to be executed in which order, learned by the strategy layer (Sec. III-B), to achieve that goal. The results regarding the strategy layer are averages of 200 experiments, in which the robot had to push an object 30 times consecutively to the goal. The confidence interval of 95% is provided. The charts regarding the skill layer are individual examples.

The strategy's state space comprised the robot's relative angle and distance to goal $g$ and object $q$:

$$(\alpha_g, d_g, \alpha_q, d_q) \in \mathbb{R}^4$$

Fig. 14 shows how the robot manages to abstract the 4-D state space into a small number of abstract regions, on which the actual strategy is learned.

The robot was equipped with only one drive as always stated. A positive lump-sum reward of 100 is given if the robot has pushed the puck to the yellow goal base. The change of the distance between the nearest puck and the goal is provided as reward rates. More formally, the motivation

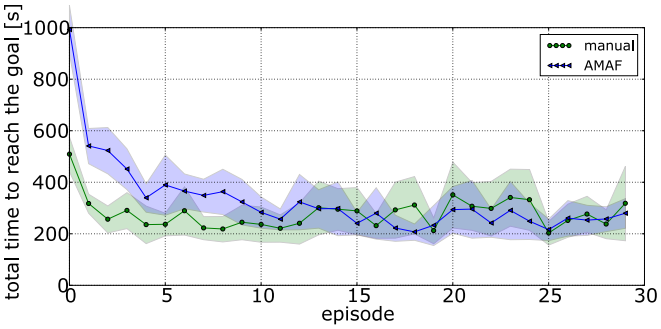


Fig. 13. Time to push the object to the goal

change was defined as follows:

$$\dot{\mu_0}(\alpha_g, d_g, \alpha_q, d_q) = \begin{cases} 100 & \text{if } d_g < 1m \\ \frac{d_q}{100} & \text{if } |\alpha_q| < 20° \\ -0.01 & \text{otherwise} \end{cases}$$

The discount parameter of the strategy was set to $\beta = 0.1$. The state space adaptation heuristics, described in Sec. III-B.3, were parameterized as follows:

- Transition heuristics: $\theta_{TV} = 0.2$.
- Experience heuristic: The number of experiences was not bounded ($\theta_M = \infty$), but stayed below 10,000 (Fig. 14).
- Failure heuristic: $\theta_f = 0.01$.
- Reward heuristic: It considered the reward rates of the last $n = 20$ interactions made in the current region and used the constants $\theta_{RV} = 0.01$ and $\theta_l = 6$.
- Simplification heuristic: An action $a$ in state $s$ was considered deterministic, if $P(a|s) > 0.8$.

The configuration settings of the skill layer are described below.

- Degrees of freedom: 2
- Controls: "decrease" ($f_c(iv_p, av_p) = |av_p|$)
- Modeling algorithms: radial basis interpolation and polynomial approximation

The robot takes more time in the first run as it also has to explore its own capabilities and learn the skills, as can be seen in Fig. 13. From an average time of $1000s$ for the first episode the time needed drops quickly to slightly more than $200s$ ("learning"). Contrasted to that the "manual" curve displays the performance, which used the same strategy layer and same configuration, but replaced the learning skill layer by a handcrafted skill set of two optimal skills. It shows that while being faster in the beginning, the learning skill layer manages to finally converge to the same performance, which is assumed nearly optimal for this scenario.

The reward per second is displayed in Fig. 15, where the learning skill layer stays slightly below the optimal, but far less robust handcrafted skill set.

This shows that ESLAS is capable of autonomously tackle infinite state and action spaces in a realistic scenarios. Although the scenario was simple it showed all the characteristics of real-world scenarios, i.e. it was noisy, continuous, and time-dependent.

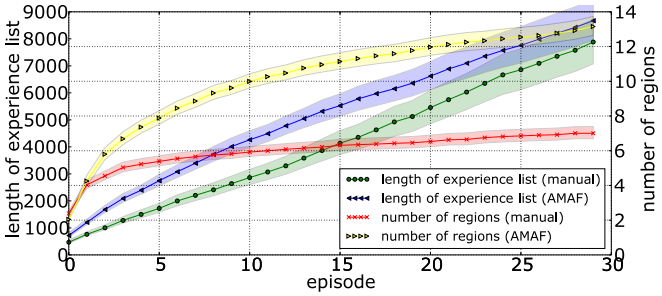


Fig. 14. Size of experience and number of abstract regions

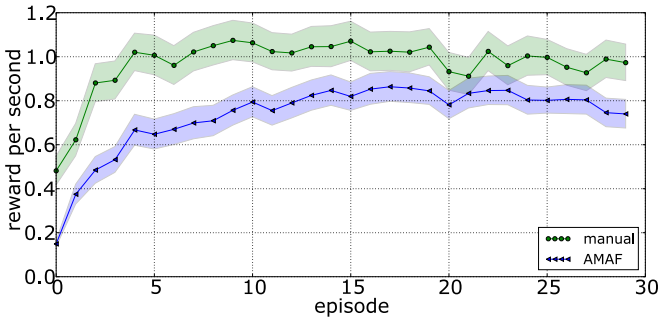


Fig. 15. The reward per second

The skill layer has autonomously generated different competing prediction models that determine the behavior of the learnt skills. We will try to represent in a synthetic way the behavior obtained by minimizing the angle to one object. We will use the prediction model based on the radial basis function approximation that predicts the next value of the angle by knowing the value of the angle and the distance to the object and the chosen low-level action. We have created a grid of 30x30 points in the input space. The input dimensions are the angle and the distance to the ball, so each point of the grid is characterized by a certain couple angle-distance. For each point of the grid, we have used the ERROR MINIMIZER to compute the low-level action that minimizes the predicted distance. The result is a pair of 3-D graphs, one indicating the chosen tangent speed and the other indicating the chosen rotation speed. We will represent the third dimension (the actuator intensity) by using colors: red for negative intensity, white for low intensity and blue for positive intensity.

In Fig. 16 the behavior of decreasing the angle to the ball is represented. The lower graph indicates the rotation speed. It is null when the angle is already minimized otherwise it is set to turn as much as possible toward the ball. The front speed is shown in the graph above and looks more confusing then the rotation speed. The only behavior that we can notice is that the robot goes back when it close to the object. In effect going on could lead to a continuous rotation around the object that would make the distance never decrease. When the distance is not low, there is not a clear behavior for what concerns the front speed. This makes sense because it does not affect much the angle to the object so it can even be chosen randomly without side effects on the performance of the action.

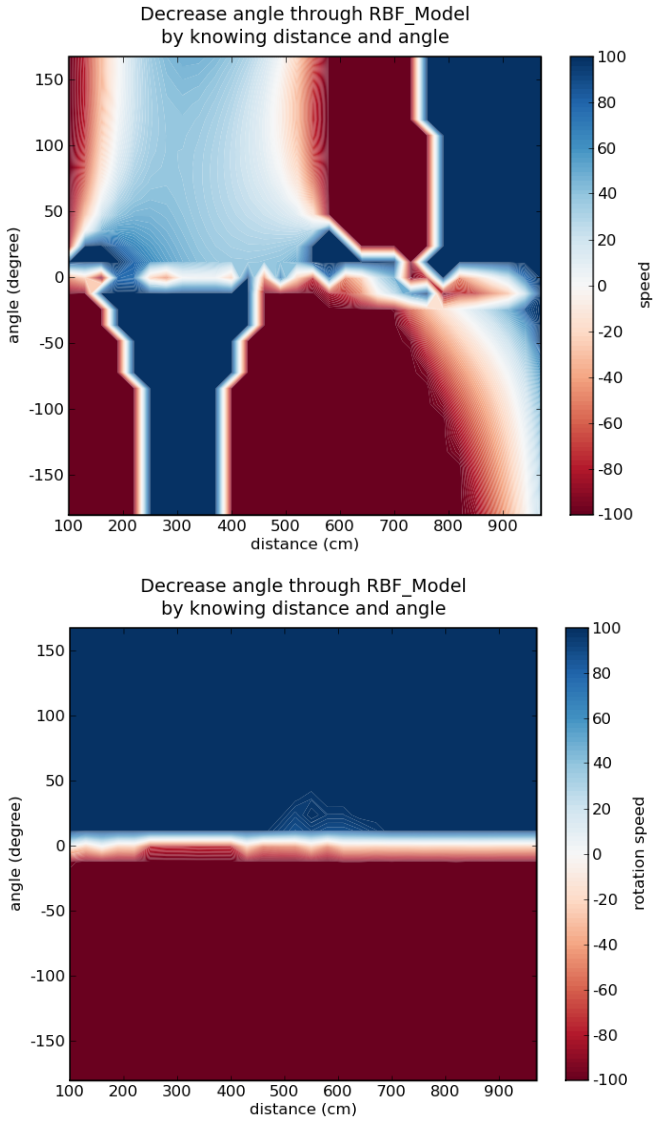


Fig. 16. Low-level actions associated to the abstract action of minimizing the angle to the ball. The red color denotes a full negative value (-100%), while the blue one a full positive one (100%).

## VI. Conclusion

In this article we presented *Evolving Societies of Learning Autonomous Systems* (ESLAS), a framework that is able to handle system and environmental changes by learning autonomously at different levels of abstraction. It is able to do so in continuous and noisy environments by 1) an active strategy-learning module that uses reinforcement learning and 2) a dynamically adapting skill module that proactively explores the robot's own action capabilities and thereby provides actions to the strategy module. We presented results that show the feasibility of simultaneously learning low-level skills and high-level strategies while both are adjusting themselves to each other. Thereby, the robot drastically increases its overall autonomy.

This architecture is not only designed for individual learning robots, but also to support imitation in multi-robot scenarios as could be shown by the authors previous work [18], [19]. In the future, the authors are planning to use the ESLAS architecture also to enable robots to cooperate even if some of them were not specifically designed to do so. This means, that the robots will be able to detect behavior patterns in the performance of robots, which are not aware of the other robots around them. These patterns are then utilize to align the observing robot's own behavior accordingly.

## References

[1] Willi Richert, Olaf Lüke, Bastian Nordmeyer, and Bernd Kleinjohann. Increasing the autonomy of mobile robots by on-line learning simultaneously at different levels of abstraction. In *International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE Computer Society, March 2008.

[2] Alexander Stoytchev. Five basic principles of developmental robotics. 2006.

[3] A. Stout, G.D Konidaris, and A.G. Barto. Intrinsically motivated reinforcement learning: A promising framework for developmental robot learning. In *The AAAI Spring Symposium on Developmental Robotics*, March 2005.

[4] H. van Hasselt and M.A. Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning (ADPRL'07)*, pages 272–279, April 2007.

[5] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, 2003.

[6] A. Lazaric, M. Restelli, and A. Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 833–840, Cambridge, MA, 2008. MIT Press.

[7] A. Bonarini, A. Lazaric, and M. Restelli. Reinforcement learning in complex environments through multiple adaptive partitions. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 531–542, 2007.

[8] M. J. Kochenderfer. *Adaptive Modelling and Planning for Learning Intelligent Behaviour*. PhD thesis, School of Informatics, University of Edinburgh, 2006.

[9] Willi Richert and Bernd Kleinjohann. Adaptivity at every layer – a modular approach for evolving societies of learning autonomous systems. In *Proceedings of IEEE/ACM ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Leipzig, Germany, 2008.

[10] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[11] A.W. Moore and C.G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993.

[12] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, 1998.

[14] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[15] Josep Call and Malinda Carpenter. Three sources of information in social learning. In K. Dautenhahn and C. Nehaniv, editors, *Imitation in animals and artifacts*, pages 211–228. MIT Press, Cambridge, MA, USA, 2002.

[16] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, Jul 2003.

[17] Russell Smith. Website of ODE (Open Dynamics Engine). http://www.ode.org/, 2008.

[18] Willi Richert, Oliver Niehörster, and Markus Koch. Layered understanding for sporadic imitation in a multi-robot scenario. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'08)*, Nice, France, 2008.

[19] Willi Richert, Ulrich Scheller, Markus Koch, Bernd Kleinjohann, and Claudius Stern. Increasing the autonomy of mobile robots by imitation in multi-robot scenarios. In *International Conference on Autonomic and Autonomous Systems (ICAS'09)*, 2009.