Self-organization at the lowest level: Proactively learning skills in autonomous systems

Willi Richert, Bernd Kleinjohann, Alexander Murmann
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany
richert@c-lab.de

Abstract: To enable autonomous systems to learn basic skills for unknown and changing environments and stay robust in case of change, Organic Computing principles have to be applied at all layers. In this work an architecture is presented that can be used at the lowest layer providing robust skills to higher-level strategy layers, that depend on encapsulated actions. With emphasis on robustness it is able to learn to control its actors without a priori information about their meaning. This is made possible by skill modules that are learned together with their action-effect dependencies and their enabling preconditions by proactively carrying out experiments within their environment. The architecture is evaluated by simulating a differentially driven robot.

1 Introduction

When some form of adaptation is needed typically a tiny part in the overall control architecture is identified and substituted by e. g. neural nets or solved by other statistical learning methods. However, they all assume a static training set, which hampers the ability to adapt appropriately to suddenly changing environments. Furthermore, they expect the designer to foresee all possible changes the system might undergo. In this work we present an architecture that is able to detect changes in the environment that render previously learned skills useless, and react in a way that relearns the obsolete parts by proactively carrying out experiments. Thereby, the designer does not have to foresee every possible change the system might undergo. With skills we understand low-level blocks of behavior that can be triggered by some higher-level strategy process. We do this by coupling learned skills with their enabling conditions that have been observed while experimenting and the effects of the action. Thereby the system can monitor progress via many fine-grained cause-effect schemata it has learned, and trigger relearning of the previously learned skill. By developing and finding basic skills the robot drastically reduces the exploration space the higher levels otherwise had to consider.

Let us assume an upper strategy layer requesting some behavior that has a certain effect on its environment. The skill learning layer then consults its skill database (Skill DB) for appropriate skill modules consisting of a set of preconditions, the action and the predicted effect (similar to *start condition*, *action type*, *end condition* in [Bis05]). If an adequate skill is found, meaning that there is some behavior that resulted in the desired effects in the past, it is executed. If not, the system starts to tinker with its environment hoping to find a concept that is connected with the goal condition and changes significantly. Then it focuses on this part and continues to experiment until it has found some model that expresses the cause-effect relationship. After some time, if the system is sufficiently confident with the learned skill it stores it in its Skill DB. This approach is supported by Gibbson's claim [Gib66] that our perception of the world is dependent on our interactions with it. For this he introduced the term *affordance* of objects meaning the possible actions that can be performed at environmental objects. If a system is able to find out which actions make sense, it has filtered out the vast amount of useless actions. We make use of this exploration space shortcut by proactively seeking for cause-effect relationships considering the governing conditions.

2 Related Work

In this work we are interested in an architecture by which an autonomous system is able to relearn completely new skills on-line. That this is generally possible has been demonstrated by Schultz, et al. with their *Continuous and Embedded Learning (CEL)* approach [SG00]. With it autonomous vehicles can adapt to partial loss of sensor capabilities online by learning a model of the environment and evolving a rule base in that simulation model. Once, sufficiently good performance has been achieved in simulation, the evolved rules are executed in the real world. Their architecture has to relearn its rule-base if a significant change in sensor capabilities has been monitored. Nording, et al. [NBB98] use *Automatic Induction of Machine Code by Genetic Programming (AIMGP)* to derive movement control code while the robot is experimenting with its environment.

All these approaches couple the lower level skills tightly to the upper strategy algorithm whether it is learning or planning. Our architecture, in contrast, encapsulates the skill learning mechanism so that it can be used by planning or learning algorithms as a black box by a predefined concept language: this architecture thus works as a skill service layer to the upper layers. This is a step towards a fully autonomic and organic system consisting of different layers, each of which can be relearned and adapted independently of the other layers. Thereby, our approach translates the properties that made e. g. the TCP/IP protocol stack so successful and robust to the autonomous system domain. It can be seen as a robust autonomous layer for learning low-level skills in a layered learning architecture, as proposed by Stone [Sto00] or in the subsumption architecture, as suggested by Brooks [Bro86]. However, we emphasize robustness in contrast to optimality, as done by Stone.

3 Architecture

For a low-level skill learning layer to support the system's robustness to a high degree it must be as independent from the other layers as possible. For this reason only a minimal interface is provided: The wanted effects are specified by the requesting layer via a behavior-request language and the final success or failure is then fed back. The behavior request language consists of user-defined concepts of its perception that define ways in which the skill learning layer can "think in". In our evaluation example, a robot navigation task, it is based on the perception, like distance or angle to recognized objects. The goal for the skill layer is to provide at every time a robust and sufficient set of skills that are guaranteed to influence the concepts in the desired way. These skills are stored in the Skill DB as shown in Fig. 1. If an effect is requested by the Exploiter and a corresponding skill

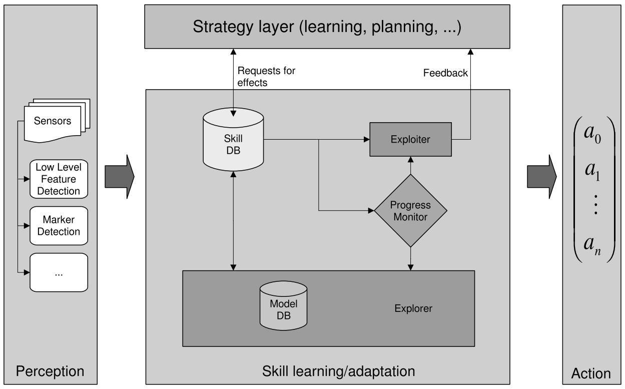

Figure 1: Autonomous skill learning architecture

module exists in the Skill DB, the system starts and carries on to execute the proper skill under the given preconditions, unless the Progress Monitor (PM) detects a deviation from the predicted outcome of the action, which triggers the adaptation process. If it is unable to fit the new data to one of the models provided by the designer (e. g. polynomials, fuzzy, ...) it delivers control over to the learning process which discards the skill in question from the Skill DB and tinkers with the environment until it has found some action setting that is able to effect the concept of the discarded skill modules (SM).

SMs are stored in the Skill DB and can be fetched according to the desired effect the robot wants to achieve at the moment and the environmental **precondition** it is currently in: $SM = (pc, a, e)$, denoting precondition, action, and effects. An **action** is a vector setting the values of the robot's actuators in the appropriate domain (e. g. for a wheel: [-1,1], for a shooting device: $\{0,1\}$, etc.). While tinkering with the environment they are collected together with the accompanying perceptions. Afterwards multiple SMs with their corresponding action vectors will be generalized to contain a parametrized action vector together with a learned function that matches the found invariant. **Effects** denote the consequences of an applied action in the predefined concept language. At first, when the system is in the tinkering phase the effect denotes a time derivative of a relevant concept. With more and more experiments it can contain arbitrary approximated functions. Currently polynomials are used for learning to control a robot with differential drive.

3.1 Behavior request language

The SM that results in the wanted behavior is requested from the skill learning layer in form of the wanted condition set G on raw perception data P: $G \subseteq \mathcal{C} \times \otimes \times \mathbb{R}$. The raw perception is projected and filtered by user-defined filter functions from the set $\mathcal{F}$: $\mathcal{C} = \{f(p) : f \in \mathcal{F}, p \in P\}$, e. g. $angle(o1)$, $sonar2()$, or $distance(o1, o2)$ for recognized objects $o1$ and $o2$. They are compared using $\otimes \in \{<, <=, =, >, >=, <>\}$. Given an existent system, the designer is able to constrain the solution space of the skill learning layer by defining $\mathcal{F}$. In addition, newly designed filters can be easily integrated at runtime, while the system is on field.

3.2 Effect

An effect describes changes to the environment from the perspective of the system as a result of an action: $e \in \mathcal{C} \times \mathcal{P}$, with $\mathcal{P}$ being the set of predictions using predefined model functions $\mathcal{M}$ the system can approximate. They are stored in the *Model DB*. The changes can be expressed in two ways: In case that the learning module successfully generalized the recorded actions and effects, the change is expressed by some model function. Otherwise, if the SM simply represents one experiment with the environment, the change is a constant: e. g. the effect *Effect(distance(o1),* -0.3*)* describes that the execution of the SM results in a decrease of distance to object $o1$ by $0.3m/s$. The effect *Effect(distance(o1),* $-0.3 \cdot a$*)* expresses that the decrease of distance is determined by some factor a, which can be used to control the action vector. The effects that are recorded in the tinkering phase will serve different purposes later on:

- The upper strategy layer will request skills from the Skill DB by providing the wanted effects as the low level goal.
- When an SM has been learned and is being executed, its effect will serve as a reference to the PM, triggering relearning of the SM when deviances are detected.

3.3 Action

An action vector has as many dimensions as the system has different actuators. The system makes no assumptions as to what purpose the different values have – this will be found out proactively. Therefore, the domains of the different actuating variables have to be provided by the system designer. Severe disturbances like e. g. randomly reassigning their meaning, can thus be handled robustly by the system in that the PM detects abnormal effects at learned SMs and will relearn them.

3.4 Precondition

Preconditions are recorded in order to detect whether an SM is applicable in the given situation. While the system is tinkering with its environment it stores all the environmental conditions. Generalizing over the preconditions is done by creating a convex hull [BDH96] of enabling conditions in $|C|$-dimensional space, $C \in \mathcal{C}$. These conditions must hold so that the according SM can be executed. In addition to the enabling precondition convex hull, a disabling hull is stored to gather information about situations in which the SM clearly is not applicable. If the Exploiter is retrieving only SMs from the Skill DB that do not match the enabling preconditions and are also not covered by the disabling preconditions, the SM is regarded as a potential skill and tried as if it were applicable. If the result is validated by the PM, the enabling hull is updated.

In addition, as the data for representing preconditions quickly gets enormous and the convex hull algorithm in general dimensional space is too time consuming to allow e. g. a robot to perform in real-time over a longer period of time, the perception is preprocessed using the non-parametrized clustering algorithm “Resource Allocating Vector Quantizer” (RAVQ) [LN00] to handle sensor information flow. RAVQ automatically builds model vectors for salient perception events which are used as salient condition description events. With this preprocessing the system in addition becomes robust to noise at the sensor level.

4 The Learning Process

While acting in its environment the robot is constantly comparing the predicted outcome of its action in the last step with the resulting situation in the current step. If then, e. g., the environment or the hardware of the robot changes due to wear out or other reasons, the PM detects the deviance of the activated SMs and has the possibility to choose the level of adaptation to the changes dependent on the severance of the changes. If this process comes to the conclusion that it is unable to adapt the parameter, it turns the control over to the learning process. It does so only for the actual skill that is executed. Thus, only the affected SMs will be adapted or relearned, and only to the minimally necessary degree. The same applies for the situation that no SM is found for the requested effect.

When the learning phase is triggered, the system has recognized that some important skill is missing or not working as expected. This is the case either because no SM in the Skill DB matches to the requested effect, or the previously executed SM turned out to have different consequences than it had when it was learned. After some time of collecting perception data with the newly chosen actor values, the data is processed by following four stages: 1) relevance checker, 2) segmentation and approximation, 3) model invariant tester, and 4) behavior generator. At first the system tries out random actor values, until the relevance checker signals a significant change in the concept of the request effect. If the concept has not changed enough, new actor values are chosen and the process is repeated. Afterwards, the time derivatives of the requested concept based on the recorded perceptions are built. To be able to generalize over the perceived data, not only a decent set of model functions $\mathcal{M}$ is needed, but also a segmentation function for every model class $m \in \mathcal{M}$, that slices the perception P into consecutive chunks of data $p_i \in P$, $\bigcup_i p_i = P$, that can be approximated by m. This is necessary because over the course of proactively experimenting in its environment the system probably perceived data that has to described in different ways. For every processed segment an individual SM will be created to account for that. The choice of the model function m is subject to future research. A straightforward approach is to just try all model functions and take the one that is able to approximate the most of the perceptual data.

For every approximated segment the invariant testers of the provided models are applied to ensure that useful abstractions has been made. The invariant testers are also dependent on the model function and detect underfitting. The system will continue to experiment and test whether its invariants match the criteria of the approximated data, until an invariant signals a sufficient match, which results in the creation of an SM using the perception data together with the action. The more sophisticated the provided models are, the less SMs will be created, because they are applicable to a wider range of situations captured as preconditions. This is a very important property for robust skills since the preconditions of the SMs will by their nature limit the range of situations in which the SM in question approximates valid behavior: the simpler the model the more local approximations will be created.

The now created SMs can be fetched the next time the system is in the exploitation phase. In this phase it requests its Skill DB for some SM with the wanted effect. If two SMs are found they can be merged into a generalized SM if the following conditions hold: 1) The action vectors are linearly dependent. 2) The preconditions are similar to a predefined degree (i. e. their convex hulls overlap). The SMs are merged by placing variables in the actor, which are then used in the effect as prediction functions. Given e. g. two skill modules b_1 and b_2 with the corresponding actors and effects. If the above mentioned conditions hold they will be discarded from the Skill DB and a new SM b will be inserted with the actor a and effect e:

$$\begin{array}{ll} a_1 = (0.2, 0.4, 0) & e_1 = Effect(Distance(100), -0.8) \\ a_2 = (0.1, 0.2, 0) & e_2 = Effect(Distance(100), -0.4) \\ a = (x, 2x, 0) & e = Effect(Distance(100), f(x)) \end{array}$$

Here, $f(x)$ is a prediction function instead of a simple prediction value. As in the current system $\mathcal{M}$ only contains polynomials we would get $f(x) = -0.4 \cdot x$. However, with the current architecture new prediction functions together with the corresponding invariant testers can quickly be added to the system.

5 Experimental Setting

We evaluated this architecture by simulation using a differentially steered Pioneer2DX robot having sonar, laser range-finders, and differential drive. The experiments were performed using the *Player/Stage* [GVH03] simulation environment. The robot was provided with three anonymous actors, of which two of them corresponded to the left and right wheel. The robot had the task to drive to a goal at the right side of the field. Thereby, a request to reach the situation $Condition(Distance(goal) < 1)$ by the upper strategy layer was simulated (Fig. 1). Every time the robot reached the goal it was repositioned randomly in the field and restarted with the previously learned skill modules (called 'run' in the following). The experiment was performed ten times with five runs each.

The system was only provided with a 1d-polynomial model function to show the overall feasibility of the approach. In that case the perception can be sliced into monotonic segments. For this quasi-monotonic intervals [FLB05] turned out to be especially suited due to their adjustable error allowances to cope with noise in the sensor readings. As invariant testers mean and variance of the fitted polynomial were used: By requesting the mean to be above some threshold and the variance below some threshold the system only approximated perceptual data that was the effect of sufficient change to the environment and enough confidence in the approximation. As this might be a too strong restriction at first view, it turned out that the architecture can pretty good cope with such simple relationships, by just generating more specific skills that map to the corresponding situation. Only if the invariant testers are passed, the perception data together with the action at that time is used to generate a skill and stored in the Skill DB.

The difficulty in the experiment was to learn the proper skills for the different situations. In contrast to omni-wheel robots, the problem with the experimental robot is that it has to learn to cope with its non-holonomicity. I. e. that in order to move, e. g., to a goal that is at the right side of it, the robot has to learn how to turn in order to be able to reach that goal. The used RAVQ parameters were $\delta = 0.3$ and $\epsilon = 2$.

6 Results

The number of simulation steps the robot needed to reach the goal at each run is shown in Fig. 2. The number of skill modules the robot learned is depicted in Fig. 3. While it needs 8,811 steps to reach the goal at the first run, the number of steps quickly drops down to 293 steps in the third. After the third run, the high CPU load causes sporadic perception errors in the robot because of which the performance slightly decreases. This is due to the increased data amount of the positive and negative convex hulls – an issue that we will pursue in the future. When bumping against a wall, the progress monitor correctly detected that something unforeseen had happened. As an effect the current situation was stored in the negative convex hull for the according skill in every time step. This is done as long as the negative convex hull is growing and current perception does not fall into it. If that is the case the SM is discontinued and will not be executed any more in the future in similar situations. As a result, another then "best" SM is chosen that leads away from the wall towards a more promising situation in which the goal can be reached. It is important to note that the number of additionally learned skills decreases with every run, as can be seen in Fig. 3. That means that the system created useful skills, which were reused over and over again, and only created new ones in previously unforeseen situations.

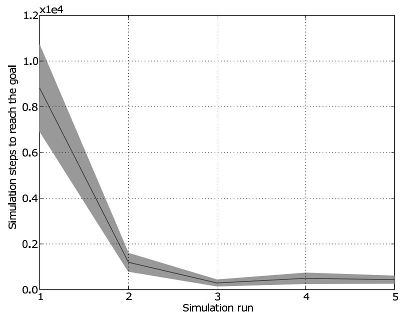

Figure 2: The number of steps needed to reach the goal condition at each run averaged over 10 trials (ordinate in thousands). The grey background is the 95% confidence interval.

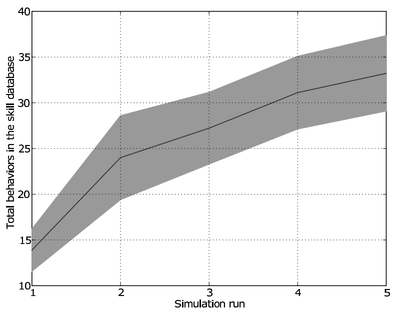

Figure 3: The number of total skills modules after each run averaged over 10 trials. The grey background is the 95% confidence interval.

7 Conclusion and Outlook

Supporting Organic Computing principles is also possible at the lowest behavior layer. The robust architecture in this paper is able to learn low-level skills guided by behavior dependent predictions it recorded while the skill was learned. Even with simple model functions the system was able to learn skills without having a priori information about its actors. As an effect of our architecture not paying attention to actor semantics and the way it dynamically creates, chooses, or deprecates skills, it has all necessary properties of being robust in unforeseen environments. This also holds if the environment or the robot itself changes due to wear out or harm caused by other robots. As long as there is a way to proceed and fulfill the goal the autonomous system will find skills to reach it.

The performance of the architecture can, however, be drastically increased in several ways: As the dimension of the state space grows proportionally with the number of concepts and number of detected features, some means is needed to generalize over the features, if more sophisticated environments are used. Instead, the system should detect the similarities and generalize them into some task dependent topology. We plan to deploy the architecture at our testbed, the mid-size soccer robots *Paderkicker* [RKK$^+$06b]. Successful experiments with online adaptation at the action sequence learning layer have shown how the knowledge transfer in societies of autonomous systems leads to the propagation of the most valuable information units that offer the biggest performance advantage [RKK05, RKK06a]. This can now be used not only to robustly learn the sequencing of behaviors, but also the behaviors themselves. We can think of a scenario where a swarm of mobile systems is working on site and only some of the mobile systems are within the designer's reach. With the architecture in this paper and a modification to the authors' previous works, the designer is able to provide some of the systems within reach with new or updated versions of $\mathcal{C}$, $\mathcal{F}$ or $\mathcal{M}$, which have the potential to increase the adaptation performance. These modules are then distributed based on the successes or failures of the individual system.

References

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[Bis05] Alexander Bisler. Organizing an Agent's Memory. In *Proceedings of the Workshop for Memory and learning mechanisms in autonomous robots at the European Conference on Artificial Life, ECAL'05*, 2005.

[Bro86] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation RA-2*, pages 14–23, 1986.

[FLB05] Will Fitzgerald, Daniel Lemire, and Martin Brooks. Quasi-Monotonic Segmentation of State Variable Behavior for Reactive Control. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 1145–1150. AAAI Press AAAI Press / The MIT Press, 2005.

[Gib66] James J. Gibson. *The Senses Considered as Perceptual Systems*. Houghton-Mifflin Company, Boston, 1966.

[GVH03] 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.

[LN00] Fredrik Linåker and Lars Niklasson. Sensory Flow Segmentation Using a Resource Allocating Vector Quantizer. In Francesc J. Ferri, José Manuel Iñesta Quereda, Adnan Amin, and Pavel Pudil, editors, *SSPR/SPR*, volume 1876 of *Lecture Notes in Computer Science*, pages 853–862. Springer, 2000.

[NBB98] Peter Nordin, Wolfgang Banzhaf, and Markus Brameier. Evolution of a world model for a miniature robot using genetic programming. *Robotics and Autonomous Systems*, 25(1-2):105–116, 1998.

[RKK05] Willi Richert, Bernd Kleinjohann, and Lisa Kleinjohann. Learning Action Sequences through Imitation in Behavior Based Architectures. In *Systems Aspects in Organic and Pervasive Computing – ARCS 2005*, number 3432 in LNCS, pages 93–107. Springer-Verlag Berlin, 14 - 17 March 2005.

[RKK06a] Willi Richert, Bernd Kleinjohann, and Lisa Kleinjohann. Trading off impact and mutation of knowledge by cooperatively learning robots. In *IFIP Conference on Biologically Inspired Cooperative Computing – BICC 2006*, 2006.

[RKK$^+$06b] Willi Richert, Bernd Kleinjohann, Markus Koch, Alexander Bruder, Stefan Rose, and Philipp Adelt. The Paderkicker Team: Autonomy in Realtime Environments. In *Proceedings of the Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, 2006.

[SG00] Alan C. Schultz and John J. Grefenstette. Continuous and Embedded Learning in Autonomous Vehicles: Adapting to Sensor Failures. In Chuck M. Shoemaker Grant R. Gerhart, Roboert W. Gunderson, editor, *Unmanned Ground Vehicle Technology II*, volume 4024 of *Proceedings of SPIE*, pages 55–62. 2000.

[Sto00] Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press, 2000.