RPL$_{\text{LEARN}}$: Extending an Autonomous Robot Control Language to Perform Experience-based Learning

Michael Beetz, Alexandra Kirsch, and Armin Müller
Lehrstuhl für Informatik IX, Technische Universität München
{*beetz, kirsch, muellear*}*@in.tum.de*

Abstract

In this paper, we extend the autonomous robot control and plan language RPL with constructs for specifying experiences, control tasks, learning systems and their parameterization, and exploration strategies. Using these constructs, the learning problems can be represented explicitly and transparently and become executable. With the extended language we rationally reconstruct parts of the AGILO *autonomous robot soccer controllers and show the feasibility and advantages of our approach.*

1. Introduction

Due to the complexity and sophistication of the skills needed in real world tasks, the development of autonomous robot controllers requires an ever increasing application of learning techniques. On the other hand, the application of learning mechanisms on their own has turned out to be not sufficient for most complex control tasks. We conclude from this situation that for challenging and complex control tasks programming and learning should be combined synergetically. In order to achieve this synergy, we embed learning capabilities into robot control by developing extensions for a robot control language that allow for the explicit specification of executable learning problems.

The need for extensions of control languages that facilitate robot learning arises from the substantial differences between *programs that learn* and *robotic agents that learn*. Learning programs are fed with training data by a teacher. In contrast, to learn effectively robotic agents must typically collect experiences by themselves. As making experiences requires the robot to take physical action and as physical action often fails or has undesired effects the experiences made by the robot might be uninformative or even obstruct learning. For example, an experience in which a robot collides with a wall and the motor stalls is at best useless for learning the robot's dynamics. Thus the effective collection of the experiences requires sophisticated control, execution monitoring, specific perception and abstraction mechanisms and therefore the collection of experiences should become part of the learning process.

The modern robot control languages we know of do neither enforce nor strongly support the rigorous design of learning mechanisms. Their transparent integration into robotic agents that learn autonomously is rather an opaque art than an engineering exercise. In this paper, we attempt to improve this situation by extending RPL, an existing control language, with constructs for specifying control tasks, experiences, learning problems, exploration strategies, etc. Using these constructs, learning problems can be represented explicitly and transparently and become executable.

As a starting point for the development of our robot learning framework we borrow concepts from experiential learning theory. Experiential learning [8] is a well researched learning methodology in adult learning, which is based on the assumption that experience plays the central role in learning. The theory defines learning as "the process whereby knowledge is created through the acquisition and transformation of experience. Knowledge results from the combination of grasping and transforming experience"[8].

In this paper, we realize these ideas by treating "experience" as a first class object in our computational model of robot learning. We view an experience as a robot's perception of its own behavior and the effects thereof in the light of specific learning tasks. Experiences in RPL$_{\textbf{LEARN}}$ allow programmers to specify mechanisms

1. for recognizing that a substream of sensory data constitutes an experience for a given learning problem,
2. for abstracting experiences given as a substream of raw sensory data into an abstract experience represented in a language suitable for learning,
3. for monitoring the collection of experiences and recognizing uninformative and misleading experiences,
4. for specifying which collections of experiences would allow for good results of the learning process, and
5. for storing and indexing collected experiences.

With the extended language we rationally reconstruct substantial parts of the controller of the AGILO autonomous

soccer robots. The reconstructed program can learn a repertoire of routines for playing competently robot soccer.

In the remainder of this paper we will describe the control language extensions needed to treat experiences and learning problems as first class objects. We will then show how these extensions enable programmers to develop learning robotic agents much more efficiently and effectively.

2. Learning Robotic Agents

As our basic conceptualization of the robotic agent and its environment, we use the *agent model* [13] in combination with the *dynamic system model* [7, 12]. The explicit representation of these models and the definition of program variables in terms of concepts of these models supports learning, planning, and failure diagnosis in robot controllers. In this paper we exploit only parts of these models, in particular observable and controllable state variables (see section 3.1). In a companion paper [10], we describe the explicit specification of additional parts of this model.

In this conceptualization, the state of the world evolves through the interaction of the controlling process — the robot's control system (or *controller*) — and the controlled process, which comprises events in the environment, physical movements of the robot and sensing operations. The purpose of the controlling process is to influence the evolution of the controlled process so that it meets the specified objectives.

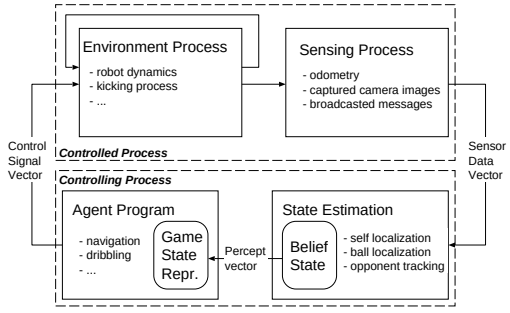

Figure 1. Dynamic system model.

Figure 1 shows a block diagram of the dynamic system model that underlies the design of our robot control system. The processes are depicted as boxes and the interactions between them as lines with arrows. There are two interactions between the controlling and the controlled process: first, the controlling process sends control signals to the controlled process to influence the evolution of the controlled process in the desired way and second, the controlling process observes the controlled process by interpreting the sensor data that are produced by the controlled process.

The agent program continually receives percepts generated by the state estimation processes and outputs control signals for the controlled process. The belief state and the control signals are explicitly represented as state variables. The state variables representing the belief state are *observable* and are automatically updated in each interpretation cycle. The state variables representing the control signals are *controllable* and their values can be asserted in the agent program. The use of state variables makes the dynamic system model in the agent program explicit and transparent.

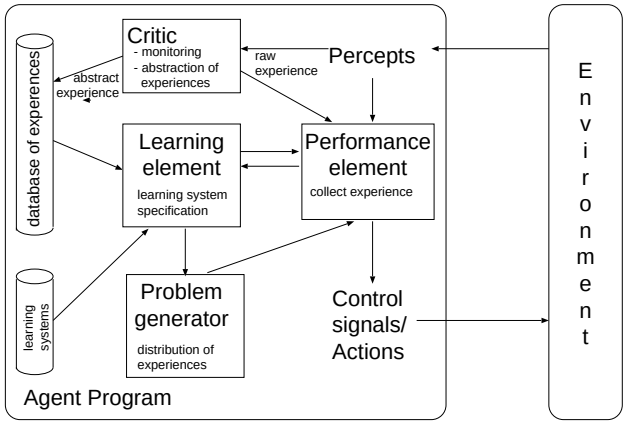

Figure 2. A general model of learning agents.

An Agent Program for a Learning Agent. Figure 2 shows a general model for learning agents according to Russell and Norvig [13] (Chapter 18). We structure the agent program of a learning robotic agent into four functional units: the performance element, the critic, the learning element, and the problem generator. The performance element specifies the agent's behavior. The critic receives the robot's percept and represents the experience in an abstract way to facilitate learning. The learning element manipulates parts of the performance element in order to improve the performance of these parts based on earlier experiences. Finally, the problem generator proposes activities to the performance element to acquire informative experiences.

Components of a Learning Robotic Agent. To get better intuitions about how control languages should be extended to support autonomous robot learning, let us look at the individual modules of the agent program in more detail.

In our computational model learning task specific experience classes are one of the most basic entities and they specify the following aspects of experiences. First, which experiences are needed to learn the task well? How can the robot make such experiences through physical action? That is, which control routines does the robot have to execute to produce them? How can the robot recognize the start and end of an experience? Given an experience as raw sensory data, is the experience informative for the respective learning task? If the experience is informative how can it be abstracted to make it more useful for learning? Finally, how

can experiences be stored and retrieved?

The *performance element* realizes the agent function, the mapping from percept sequences into the actions that should be performed next. To facilitate learning the parts of the performance element that should be learned and adapted should be represented in such a way that the learning element can reason about, modify, and generate them. The performance element should also provide means to collect data, which also monitors the data collection process and steps in when failures and problems occur. In addition, precautions have to be taken if functions and control routines haven't been learned yet.

The *critic* is best thought of as a learning task specific abstract sensor that transforms raw sensor data into information relevant for the learning element. To do so the critic monitors the collection of experiences and assesses whether a given episode is informative for the learning task. The informative episodes are abstracted into a feature representation that facilitates learning. The critic also generates feedback signals or rewards that assess the robot's performance during an episode. Finally, the episodes are stored and maintained as resources for learning.

The *learning element* applies different learning techniques such as neural network learning or decision tree learning. The learning element also specifies the appropriate parameterization of the learning mechanism, the bias, to perform the learning task effectively. Finally, the learning element specifies how the result of the learning process is to be transformed into a piece of code that can be executed by the performance element.

The *problem generator* can be called with an experience class and returns a new parameterization for the routine that collects the experiences. The new parameterizations are generated as specified in the distribution of parameterizations of the experience class.

The Interpretation Cycle. The basic interpretation cycle of the learning agent's program operates in two modes: active and passive. In the passive mode the agent performs the actions proposed by the agent function. Concurrently the critic observes the sensor stream to detect the start and end of episodes. If the recognized episodes are assessed to be informative they are abstracted and then stored into the episode database. In the active mode the critic operates in the same way. The performance element carries out a loop in which it asks the problem generator for the next parameterization of the experience collecting routine. It then executes the routine until completion and asks for the next one.

Discussion. To perform a particular learning task competently, autonomously, and efficiently all four components of the learning agent have to be tailored for the learning task. We propose to extend the control languages to provide the means for specifying learning task specific performance elements, critics, learning elements, and problem generators.

An example problem for experience-based learning. As our running example for the rest of the paper we will take a simple control task that we have solved for the AGILO autonomous soccer robots [5]. Our hardware platforms are pioneer I robots with a standard color CCD camera as their primary sensing device. The AGILO robots use a sophisticated probabilistic state estimation mechanism to estimate the robot's global position on the football field both reliably and accurately [14]. Thus in our experimental setting the robot's agent program gets the most probable robot position and orientation with respect to the global football field coordinate system and the robot's rotational and translational velocity in its percept vector.

Pioneer I robots have a simple differential drive that is controlled by specifying the speed at which each of the two drive wheels are to turn. The relative speed of the two drive wheels determines the radius of the curve the robot moves along and the absolute value of the robot's speed. We use an abstract interface that allows for the drive control in terms of a desired rotational and translational velocity of the robot. Thus our control signal vector is a pair of the desired rotational and translational velocity. Steering differential drives for complex navigation tasks with high performance is very difficult and therefore we will learn the steering routine from experience. As described and justified by experimental results in our earlier work [6] we perform our learning tasks in a simulator with the robot dynamics learned from the real physical robots.

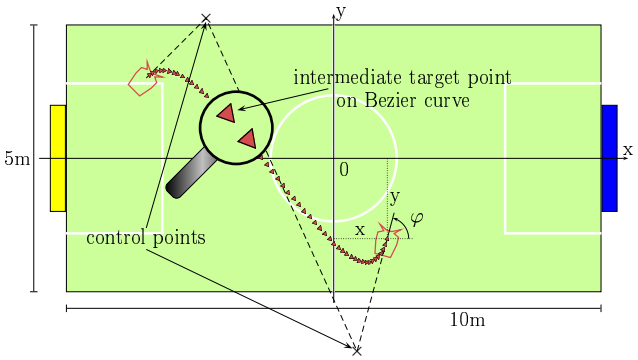

Figure 3. Visualization of state variables

As the control task we consider the task of navigating from a given start pose (position and orientation) of the robot to a goal pose. According to our objective of combining programming with learning we solve the navigation task using a routine that determines the shape of the path through programmed heuristic methods combined with learned subtasks for trajectory following. To determine the shape of the path we have heuristic rules for setting control points and compute a Bezier curve from the start to the end state constrained by the control points. The programmed part selects an intermediate target point on the Bezier curve that is vis-

ible and moderately close. This process is shown in figure 3, where the intermediate points of the Bezier curve are depicted as triangles.

The routines for approaching the next intermediate target point are to be acquired through experience-based learning. The basic idea for learning the approach routine is to let the robot drive all kinds of simple curves: narrow curves and wide curves and those with different velocities. We then select for the simple navigation tasks the most efficient experiences and use these experiences to train a neural network to be used as the intermediate target approach routine.

3. RPL$_{LEARN}$

For the implementation of the agent program we use the reactive robot control/plan language RPL [9], which we will extend to support experience-based learning. RPL has been successfully used by high-level controllers of real-world robotic agents including autonomous tourguide robots [3] and robot office courier applications [4].

RPL (Reactive Plan Language) [9] is a very expressive high-level robot control language. It provides conditionals, loops, program variables, processes, and subroutines as well as high-level constructs (interrupts, monitors) for synchronizing parallel actions. To make plans reactive and robust, it incorporates sensing and monitoring actions, and reactions triggered by observed events.

There are two more features that make RPL suitable for the implementation of learning robotic agents. First, the RPL programs are represented at execution time as plans: control programs that cannot only be executed but also reasoned about and modified. This feature facilitates the operation of the learning element by explicitly representing the code pieces to be learned and adapted. Second, RPL provides a macro mechanism that we use to introduce new control structures needed for experience-based learning.

3.1. Specifying the Dynamic System Model

A common problem in the robot controllers is the inconsistent naming of program variables that store physical quantities. Thus, the first extension to RPL that we make is not specific to learning but equally useful in many other aspects of autonomous robot control: we require that programmers explicitly specify state variables and their physical meanings.

Below comes the specification of some state variables, the percept and control signal vector. Consider our dynamic system model depicted in figure 1 for reference. State variables are represented in the control system as representational units with the following components: the *name* of the state variable which has to be unique within the control system, the physical *meaning* of its values, its *orientation*, i.e., if the variable denotes a vector it specifies the direction of the vector, and the *unit of measurement* of the variable values and a declaration of whether the variable is observable and/or controllable.

```
declare state vars
   x                  position(robot)   dim: ⟨x⟩                    unit: m                observable
   y                  position(robot)   dim: ⟨y⟩                    unit: m                observable
   φ                  position(robot)   dim: ⟨φ⟩                    unit: degrees          observable
   position           position(robot)   dim: ⟨x, y⟩                 unit: m, m             observable
   pose               position(robot)   dim: ⟨x, y, φ⟩               unit: m, m, degrees    observable
   v_translation      velocity(robot)   dim: ⟨v_translation⟩         unit: m/s              observable
   v_rotation         velocity(robot)   dim: ⟨v_rotation⟩            unit: degrees/s        observable
   c_translation      command(robot)    dim: ⟨c_translation⟩         unit: m/s              controllable
   c_rotation         command(robot)    dim: ⟨c_rotation⟩            unit: degrees/s        controllable
```

Using the state variables declared above we can then declare the percept and control signal vectors:

```
declare percept vector ⟨x, y, φ, v_translation, v_rotation⟩
declare control signal vector ⟨c_translation, c_rotation⟩
```

These specifications are much more than a more rigorous form of documentation and the introduction of consistent naming conventions. They are used for automatically writing and parsing log files, for automatically specifying database table schemata, and other tedious bookkeeping work. The specifications will become even more important as our reasoning tasks about control programs become more complex.

3.2. Specifying an Experience

In sections 1 and 2 we have motivated and outlined the pieces of information that have to be provided by a programmer to realize robotic agents that perform experience-based learning. In this section we will describe the language constructs that we introduce into RPL to specify these pieces of information explicitly and transparently. We will start with experiences.

Experience classes are defined using the RPL**LEARN** macro def-experience-class. The parameter ⟨*learning tasks*⟩ specifies the set of learning tasks where this class of experiences can be used as examples to learn from. The next parameter ⟨*feature language*⟩ describes the abstract parameters in which experiences of the class are represented for better learning performance. The specification of the abstraction itself is then indicated by the keyword abstraction and is a tuple or a mapping from one tuple into another one. Each tuple element must be either a state variable or a feature of the abstract feature language. The distribution of experiences that should be acquired through active collection of experiences is then specified as the parameter ⟨*distribution*⟩. In addition, the programmer has to provide methods for the recognition and the active acquisition of experiences. Two more methods are generated automatically from the definition of the abstract experience: the method for abstracting raw experiences and the method for storing episodes in the episode database.

```
def-experience-class nav experience
   learning tasks ⟨learning tasks⟩
   with-feature-language ⟨feature language⟩
      abstraction ⟨abstraction⟩
      distribution ⟨distribution⟩
      methods ⟨detect-method⟩, ⟨collect-method⟩
```

To show how the different parameters of the experience class are specified we use the simple learning task we have specified in section 2.

Let us first specify the episode recognition routine. Essentially this method defines the conditions that indicate that an episode starts, terminates, and that the physical action has failed. In the example below the method is parameterized with the pose that the robot should start with and the orientation of the robot that should terminate the experience. Thus the experience should start when the robot has reached the starting position and end if the orientation has reached the value φ_{end}. We have two kinds of failures: the robot might not finish the experience within the given time resources and second the robot moved out of bounds. In the first case the collection process is aborted in the second case it is to be retried with a different starting position.

```
method detect nav experiences (x_start, y_start, φ_start, φ_end)
  start-cond  x = x_start ∧ y = y_start ∧ φ = φ_start
  end-cond    φ = φ_end
  failure-conds  if |x| > fieldsize_x  throw out-of-bounds
                 if time > t_max       throw time-out
  failure-handlers  catch out-of-bounds do restart
                    catch time-out      do abort
```

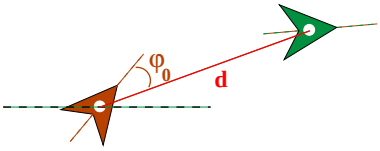

Figure 4. Visualization of features

As the next component of our experience class we specify the language for abstracting experiences. Here, we name new features and specify how they are to be computed from the state variables. The features we have defined for our example problem are illustrated in figure 4.

```
define feature language  nav feature language
  features
    d ← √((x_goal − x_current)² + (y_goal − y_current)²)
    φ0 ← | φ_current − arctan( (x_goal − x_current) / (y_goal − y_current) ) |
```

The abstract experience is then a tuple that contains only state variables and features from the feature language. In our case the abstraction is

```
abstraction  nav abstraction
  d × φ0 → c_translation × c_rotation
```

The collect method is a piece of an RPL plan that generates the physical actions needed for the collection of a single experience. Given the starting position and the desired rotational and translational velocities of the robot as parameters the RPL plan first navigates to the starting position and then moves with the specified velocities.

```
method  collect nav experiences
  rpl-procedure
    seq  ⟨setup-rpl-proc⟩
         ⟨control-rpl-proc⟩
```

Finally, we specify the distribution of the experiences to be collected for learning our control task. It is often useful to specify more than one distribution, in order to try several combinations of experiences later on. Therefore we define the relevant parameters for the distribution first.

```
specify distribution parameters  nav distribution
    ⟨x, y, φ⟩start:  constant ⟨-5.0, -2.5, 0.0⟩
    φend:            constant 90.0
    rotation:        range ⟨1.0, 180.0⟩
    translation:     range ⟨0.0, 1.0⟩
```

Now we can define different distributions by setting the values of the non-constant parameters. The values of a parameters can be obtained systematically, randomly or by a list of fixed values. If not stated otherwise, the parameters are assumed to be independent, thus the overall distribution is the cross product of their values. We can however specify distributions over combinations of parameters as well.

```
declare distribution  medium curves of type  nav distribution
    rotation:    systematic range ⟨20.0, 60.0⟩ step 1.0
    translation: systematic range ⟨0.2, 1.0⟩ step 0.05
```

3.3. Specifying Control and Learning Tasks

In the last section we have described how the robot can make the experiences that it needs for learning. Now we look at control tasks, entities in our control system that can be learned and at learning elements, the elements that perform the learning tasks. A *control task* is the representation of a skill the robot should possess, for example going to the ball or scoring a goal. Control tasks can be realized in different ways — the *control routines*.

In our running example we have already encountered a small hierarchy of control tasks and routines. The control task of navigation with a specified goal orientation is specialized into a navigation routine using Bezier curves (figure 3). For following the Bezier curve we introduce a second control task, whose specification is given below.

```
control task  navigate without orientation
  goal specification  x, y
  control process specification
    achievable(⟨x, y⟩) ⇒ active
    ¬achievable(⟨x, y⟩) ⇒ fail
    ⟨x, y⟩ = ⟨x_goal, y_goal⟩ ⇒ succeed
  control routines  variant 1, variant 2, variant 3
```

We have simplified the navigation task by dropping the goal orientation. Furthermore we specify the failure and success conditions of the task. For the simple navigation task the robot will learn three alternative control routines, which are described in section 4. To do so, we have to specify a learning problem.

```
learning problem  variant 1
  experiences  nav experiences 1
  learning element  nav learning element
```

A learning problem consists of two parts: the experiences and a learning element. In our example the learning element remains the same for all three control routines, only the experiences are altered. The learning element uses the

learning system SNNS (Stuttgart Neural Network Simulator). The number of input and output nodes is deduced from the abstraction specified in section 3.2. The remaining parameters are set by the user as follows:

```
learning element  nav learning element
   use system  SNNS
   with parameters
        hidden units:       5
        cycles:             50
        learning function: Rprop
```

4. Learning Navigation Routines

After having introduced our language extensions we will now show how the constructs are transformed into executable code and that the constructs make indeed important aspects of the learning tasks transparent and programmable.

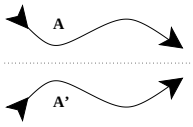

Figure 5. Exploited symmetries

To illustrate the advantages of specifying the learning problem explicitly, we compare three variants of the learning problem introduced in section 2. The variants use the same mechanisms for collecting experiences and learning, but differ with respect to the experience sets they use. The experience sets are varied along two dimensions: the experience abstraction and the distribution. One of the abstractions uses a reflection factor that exploits symmetries with respect to reflections along the x-axis, while the other one does not (as illustrated in figure 5). Our second distribution is a subset of the first one omitting examples with low translational velocity. Thus, the distribution in section 3.2 is replaced by the following one:

```
declare distribution  medium curves fast of type  nav distribution
     rotation:    systematic  range  ⟨20.0, 60.0⟩ step  1.0
     translation: systematic  range  ⟨0.6, 1.0⟩ step  0.05
```

So we have three variants of experience sets. The first doesn't exploit any symmetry-axis and uses the original distribution. In the second variant symmetries are exploited by the abstraction and the distribution remains as in the first variant. For further improvement the third variant uses a distribution with fewer slow experiences.

Learning Steps. The first step in a learning process is the acquisition of experiences. The ⟨*detect method*⟩ specifies that part of the critic that monitors the collection of data. The ⟨*detect method*⟩ is translated into a piece of RPL code that is wrapped around the control program. The wrapped code is a monitoring procedure that runs concurrently with the active program. The monitor waits until the starting condition of the experience becomes true. Then the monitor starts recording the experience and waits for the end condition of the experience. After the end the raw experience is stored into the episode database. An excerpt of the experience log is shown below.

```
(start-raw-exp :EXP-CLASS nav-experience :EXP-ID 5)
(Percept :T 1 :X -4.3 :Y -2.4 :φ 0.85 :V_tr 0.73 :V_rot 0.07)
(Command :TRANSLATION 0.75 :ROTATION 17.0 :KICK ⊥
(Percept :T 2 :X -4.25 :Y -2.49 :φ 0.96 :V_tr 0.72 :V_rot 2.17)
(Command :TRANSLATION 0.75 :ROTATION 17.0 :KICK ⊥
...
(Command :TRANSLATION 0.75 :ROTATION 17.0 :KICK ⊥)
(Percept :T 8 :X -3.83 :Y -2.48 :φ 2.19 :V_tr 0.72 :V_rot 2.04)
(end-raw-exp :SUCCESS)
```

If the critic classifies the experience as informative (in our case it does, because the collection has succeeded), it is accepted as a raw experience and stored in the experience data base by the procedure ⟨*store method*⟩. The next step is to transform raw experiences into abstract ones. The ⟨*abstract method*⟩ for performing the transformation is generated automatically from the given feature language. The database entry of abstract experiences extracted from the raw experience given above would look like this:

abstract-navigation-experience				
id	d	φ_0	$c_{translation}$	$c_{rotation}$
1	0.071310	131.981	0.75	17.0
2	0.142221	132.061	0.75	17.0
3	0.070911	125.679	0.75	17.0
...				

Figure 6 shows the resulting distribution of experiences. Having acquired the experiences the robot starts the actual learning process. This is simply done by calling a ⟨*learn method*⟩, which is provided by the learning system, using the parameters given in the learning task specification. The learn method also generates an executable function that can be called by the robot's control program.

Results. The three simple navigation routines were learned automatically and then used to navigate by following Bezier curves. The quality of the navigation routines was measured in terms of accuracy and speed. Figure 8 (a) shows the derivation of the goal angle. To compare the time needed to reach a goal position, the derivation from the average time of the three variants is depicted in figure 8 (b). Higher values denote less time for a navigation task.

Apart from the statistical evaluation we plotted the course of the robot using each of the navigation routines

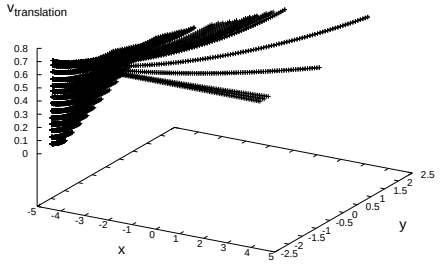

Figure 6. Specified experience distribution.

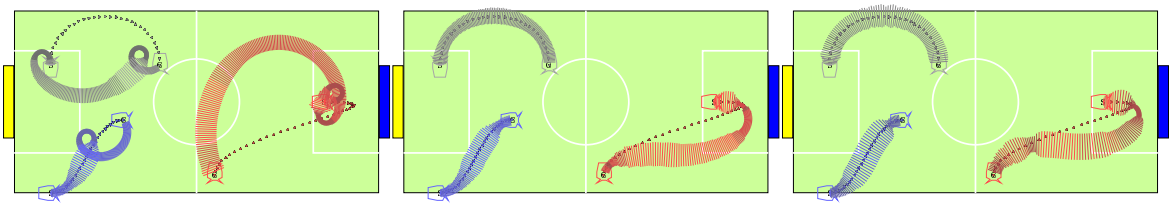

Figure 7. Performance of navigation routine: (a) Variant 1; (b) Variant 2; (c) Variant 3

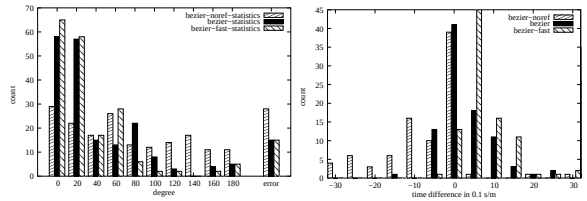

Figure 8. (a) Accuracy; (b) Time Statistics

in order to see possible deficits (figure 7). The little triangles depict the Bezier curve that is to be followed, the robot's translational velocity is represented by the length of a line orthogonal to the orientation of the robot and the rotational velocity is shown by the tint of the colors. It is obvious from all three methods of evaluation that the first variant does very badly. The routine hardly follows the Bezier curve. The goal position is often reached, the orientation however is reached by mere chance. The navigation is also very slow, because of long detours. The small alteration in the state space of the second variant has a considerable effect on the performance. This is not surprising, since the state space is cut in half by exploiting reflection invariances. The Bezier curve is usually followed easily, wherefore the goal position and orientation are reached without problems. With the transition to the third variant the enhancement of the performance is not as drastic, but still an improvement is perceptible. The goal is usually reached faster with comparable, sometimes slightly better, accuracy.

Discussion. There are several advantages of the approach we propose. Since experience distributions are objects we can maintain alternative distributions concurrently during the development of the controller. In particular we can generate distributions from the log data of games and thereby for example acquire opponent specific experience distributions. Another big advantage is the storage of collected experiences in databases. We can selectively query specializations of navigation tasks and learn specific navigation routines or we can delete experiences where the robot didn't perform well. The same holds for the representation of different feature languages. Another remark we would like to make is that our experience-based learning controllers are fully operational within the AGILO controllers.

5. Related Work

Several programming languages have been proposed and extended to provide learning capabilities. Thrun [17] has proposed CES, a C++ software library that provides probabilistic inference mechanisms and function approximators. Unlike our approach a main objective of CES is the compact implementation of robot controllers. CLIP/CLASP [1] is a macro extension of LISP, which supports the collection of experimental data and its empirical analysis. Programmable Reinforcement Learning Agents [2] is a language that combines reinforcement learning with constructs from programming languages such as loops, parameterization, aborts, interrupts, and memory variables. This leads to a full expressive programming language, which allows designers to elegantly integrate actions that are constrained using prior knowledge with actions that have to be learned. None of these projects addresses the problem of acquiring and selecting the data used for learning. This leads to a poorer performance of the learning process.

More complex tasks have been dealt with in the RoboCup domain. In the simulation league, reinforcement learning techniques are being scaled to deal with the much larger state spaces. An example is [15], which uses SARSA-learning and linear tile-coding, along with various adaptations such as predefined hand-coded skills and a reduction in the number of players, to learn a Keep-away task. Work described in [16] focuses on stronger integration of control and perception, with a hierarchical learning approach, applied to the complex tasks in the middle-size league. This integration leads to good results. Although both works solve complex learning tasks, they do not integrate the learning mechanisms into the programs.

Williams [11] has applied model-based reasoning techniques to the control of an autonomous spacecraft. In his case the models are component models of the electrical system where the system interactions are relatively fixed and known a priori. As the applications he realizes are high-risk and have high reliability constraints, learning of control routines is not extensively investigated in this approach.

6. Conclusions

In this paper, we have extended the reactive plan language RPL with constructs that support experience-based robot learning. In the extended language entities such as experience classes, control tasks, learning problems, and data collection strategies can be represented explicitly and transparently, and made executable. In the learning and execution phase of the extended control language, these entities are first class objects that control programs cannot only execute but also reason about and manipulate. These capabilities enable robot learning systems to dynamically reorganize state spaces and to incorporate user advice into the formulation of learning problems. We have also shown how the constructs are made executable and that the adequate specification of these entities gives us very powerful mechanisms for the realization of high performance robot learning systems. The extensions that we have presented are expressive enough to rationally reconstruct substantial parts of an existing autonomous robot soccer control system — the AGILO robot controller.

In this paper we have presented preliminary results and addressed only a restricted scope of learning mechanisms. Additional complex control systems need to be implemented using our approach and the conciseness and expressivity of our constructs need to be assessed and analyzed. We are just starting to incorporate optimizing learning techniques such as reinforcement learning into our approach.

We see the main impact of our framework along two important dimensions. From a software engineering perspective, the language extensions allow for transparent implementation of learning steps and abstract representation of complex physical systems. These aspects are typically not adequately addressed in current control systems, which makes them hard to understand and adapt to new requirements and conditions. The second dimension, which we find much more exciting, is the use of the framework as a tool for investigating more general and powerful computational models of autonomous robot learning. The programmability of learning systems, the modifiability of state representations, the possibility of reparameterizing learning systems, and the executability of learning specifications within the framework enables us to solve complex robot learning tasks by automatic programs without human interaction. The framework thereby enables us to investigate adaptive robot control systems that can autonomously acquire very sophisticated skills and competent task control mechanisms for a variety of performance tasks.

References

[1] S. Anderson, D. Hart, J. Westbrook, and P. Cohen. A toolbox for analyzing programs. *International Journal of Artificial Intelligence Tools*, 4(1):257–279, 1995.

[2] D. Andre and S. Russell. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, 2001. MIT Press.

[3] M. Beetz. Structured reactive controllers for service robots in human working environments. In G. Kraetzschmar and G. Palm, editors, *Hybrid Information Processing in Adaptive Autonomous Vehicles*. Springer, 1998.

[4] M. Beetz, T. Arbuckle, M. Bennewitz, W. Burgard, A. Cremers, D. Fox, H. Grosskreutz, D. Hähnel, and D. Schulz. Integrated plan-based control of autonomous service robots in human environments. *IEEE Intelligent Systems*, 2001.

[5] M. Beetz, S. Buck, R. Hanek, T. Schmitt, and B. Radig. The AGILO autonomous robot soccer team: Computational principles, experiences, and perspectives. In *International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS) 2002*, pages 805–812, Bologna, Italy, 2002.

[6] M. Beetz, T. Schmitt, R. Hanek, S. Buck, F. Stulp, D. Schröter, and B. Radig. The AGILO robot soccer team - experience-based learning and probabilistic reasoning in autonomous robot control. *Autonomous Robots*, 2004.

[7] T. Dean and M. Wellmann. *Planning and Control*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[8] D. Kolb. *Experiential Learning: Experience as the Source of Learning and Development*. Financial Times Prentice Hall, New York, NY, 1984.

[9] D. McDermott. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University, 1991.

[10] A. Müller, A. Kirsch, and M. Beetz. Object-oriented model-based extensions of robot control languages. submitted to German Conference on Artificial Intelligence, 2004.

[11] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[12] K. Passino and P. Antsaklis. A system and control-theoretic perspective on artificial intelligence planning systems. *Applied Artificial Intelligence*, 3:1–32, 1989.

[13] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[14] T. Schmitt, R. Hanek, M. Beetz, S. Buck, and B. Radig. Cooperative probabilistic state estimation for vision-based autonomous mobile robots. *IEEE Transactions on Robotics and Automation*, 18(5), October 2002.

[15] P. Stone and R. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proc. 18th International Conf. on Machine Learning*. Morgan Kaufmann, 2001.

[16] Y. Takahashi and M. Asada. Multi-controller fusion in multi-layered reinforcement learning. In *International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI2001)*, pages 7–12, 2001.

[17] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, 2000. IEEE.