Making Robot Learning Controllable: A Case Study in Robot Navigation

Alexandra Kirsch, Michael Schweitzer, Michael Beetz

Abstract

In many applications the performance of learned robot controllers drags behind those of the respective hand-coded ones. In our view, this situation is caused not mainly by deficiencies of the learning algorithms but rather by an insufficient embedding of learning in robot control programs. This paper presents a case study in which ROLL, a robot control language that allows for explicit representations of learning problems, is applied to learning robot navigation tasks. The case study shows that ROLL's constructs for specifying learning problems (1) make aspects of autonomous robot learning explicit and controllable; (2) have an enormous impact on the performance of the learned controllers and therefore encourage the engineering of high performance learners; (3) make the learning processes repeatable and allow for writing bootstrapping robot controllers. Taken together the approach constitutes an important step towards *engineering* controllers of autonomous learning robots.

Introduction

Implementing competent autonomous robot control systems that can accomplish large spectra of dynamically changing and interacting tasks is very difficult. The realization and maintenance during their development requires the control systems to be equipped with, and make ample use of autonomous learning mechanisms. Unfortunately, the performance of learned routines typically drags substantially behind the performance of handcoded ones, at least if the control tasks are complex, interact, and are dynamically changing.

In our view, this situation is caused not primarily by deficiencies of learning algorithms but rather by an insufficient embedding of learning into robot control. For a robot to learn effectively and successfully it does not suffice to merely apply the right learning algorithm. Rather, the robot must also be able to recognize the experiences relevant for learning, to actively acquire specific experiences to accelerate learning and to select informative experiences and throw away misleading ones. We know from data mining applications that the realizations of these tasks have an tremendous impact on the performance of learning and data mining applications.

Beetz et al. [2004] have proposed ROLL (Robot Learning Language, formerly called $\text{RPL}_{\text{LEARN}}$), an extension to the robot control and plan language RPL, that allows programmers to specify such mechanisms declaratively and modularily as part of the control program. ROLL introduces experiences, distributions and abstractions thereof, learning tasks, and learned routines as first class objects into the language. It also provides transparent and modular specification mechanisms for them. Using ROLL, a programmer can specify an experience class relevant for a given learning task by adding a perception mechanism for it, a routine for performing physical actions to gather such experiences, a critic that decides whether or not an experience is informative. Using the last specification, a robot can recognize an experience in which it collided with an object, which is not informative for learning the dynamics of the robot.

In this paper we evaluate some of the claims made by Beetz et al. [2004] by applying the extended language to learning an example class of navigation tasks for autonomous robot soccer. In the context of this case study, we will discuss whether changing the parameters and the mechanisms that are made explicit in ROLL have substantial impact on the performance of learned routines. We will also investigate whether the changes can be made modularily and transparently and whether the language extensions encourage the *engineering* of autonomously learning controllers through a seamless integration of programming and learning.

The case study demonstrates the huge potential of control languages that support learning for the realization of more competent controllers that are easier to develop and maintain.

In the remainder of this paper we proceed as follows. Section briefly introduces some of the language constructs provided by ROLL that are used for our case study. In section we present a case study demonstrating the language ROLL. Thereafter we compare several navigation routines that were implemented with ROLL. Finally we discuss related work and conclude with section .

The Robot Learning Language ROLL

Before we start with our case study, let us first describe the computational model for the interpretation of ROLL controllers and then the key language constructs provided by ROLL for the embedding of learning mechanisms.

The Interpretation Model of RoLL

The extended robot control language RoLL assumes that RoLL controllers are executed by an interpretation model that has the structure and components depicted in figure 1. The main parts of the system are the *performance element* that controls the robot, the *critic* that executes the learning task specific perception mechanisms, the *learning element* that reasons about and modifies the performance element in order to improve its behavior. To do so, the learning element uses a database of experiences and a library of learning algorithms as its resources. Finally, the computational model includes a *problem generator* that allows the robot to acquire relevant experiences actively. In the remainder of this section we will briefly sketch the functionality of the individual components of the interpretation model.

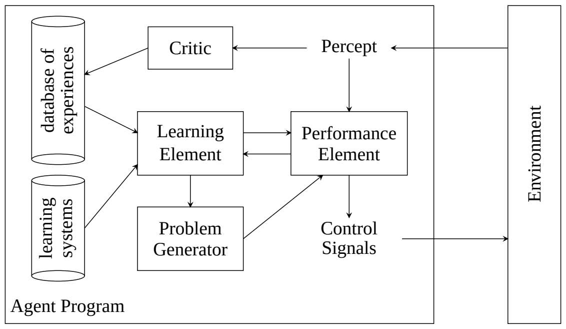


Figure 1: Overview of a learning agent after (Russell & Norvig 1995)

The *performance element* realizes the mapping from percepts into the actions that should be performed next. It contains code pieces, called *control tasks* that might not yet be executable or optimized. These are the code pieces to be learned. Thus the RoLL interpreter might have to interpret a control task that has not yet been learned, using the RoLL specification of the learning problem. In this case, it automatically activates the learning process including the collection of the necessary experiences, and continues with the interpretation after the learning process has generated an executable code piece for the control task.

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 abstracts them 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 in a relational database system coupled with a datamining toolset as resources for learning. The current RoLL version uses MySQL[1] as its database system and Weka[2] for data mining.

The *learning element* uses experiences made by the robot in order to learn the routine for the given control task. To do so, the learning element selects a subset of experiences from the episode database and transforms these experiences into input data for the learning algorithm to be applied. 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* is called with an experience class and returns a control routine that, when executed, will generate an experience of the respective class. The new parameterizations are generated as specified in the distribution of parameterizations of the experience class.

[1]http://www.mysql.com/

[2]http://sourceforge.net/projects/weka/

Learning-specific Constructs of RoLL

In order to write a RoLL controller, a programmer has to specify control tasks that are to be learned, experiences that are needed to learn a routine for the tasks, abstractions of experiences that are better correlated with the concepts to be learned, and learning algorithms, their parameterization, conversions of experiences into input data of the algorithm and the transformations of the algorithm output into pieces of the control program. For each of these aspects RoLL provides modular and transparent means for their specification.

To specify the experiences for a learning task we must code how the experiences are to be recognized, how they can be actively acquired by performing control routines, and what the distribution of experiences should be. The performance of learned routines often improves as the distribution of experiences matches the expected distribution of control tasks which they are learned for. To facilitate learning the programmer can also define suitable abstractions or "feature languages". The experiences are stored in an episode database automatically.

```
experience class ⟨name⟩
  with feature language ⟨feature language⟩
    abstraction ⟨abstraction⟩
    distribution ⟨distribution⟩
    methods ⟨detect-method⟩, ⟨collect-method⟩
```

A learning problem consists of two parts: the experiences and a learning element. The experiences are extracted from a database. This gives the programmer the freedom to choose from the gathered experiences only those that are most suited for the particular problem. For this purpose we use an abstract language that was designed for data cleaning (Galhardas *et al.* 2001). This language is an extension of SQL and provides, among others, constructs for matching, clustering, and merging of data. Thus a set of experiences of an experience class can be used for different learning problems. The learning element contains the choice of a learning algorithms and its parameterization for the learning problem.

```
learning problem ⟨name⟩
  experiences ⟨experience set⟩
  learning element ⟨learning element⟩
```

Given a set of experiences a robot learning problem is essentially the application of an appropriately parameterized learning algorithm, the transformation of the abstracted experiences into the input format of the learning algorithm, and the generation of code that is executable within the controller and solves the control task from the output of the algorithm.

Navigation with Bézier Curves: A Case Study

In the domain of robot soccer, the navigation is a fundamental issue. We consider mobile robots with a simple differential drive which we can control with an an abstract interface that allows for the drive control in terms of a desired rotational velocity v and translational velocity ω of the robot. Steering differential drives for complex navigation tasks with high performance is very difficult. As described and justified by experimental results in (Beetz *et al.* 2004) we perform learning tasks in a simulator with the robot dynamics learned from the real physical robots.

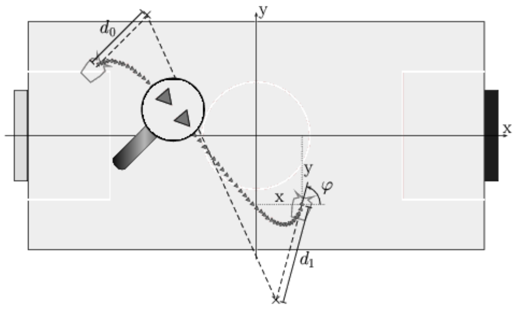


Figure 2: The navigation problem using Bézier curves

In robot soccer it often does not suffice to reach a position, but positions must be reached in orientations that facilitate subsequent actions. Thus we consider navigation tasks that are specified by the current robot position and orientation $\langle x, y, \varphi\rangle$ and the intended one $\langle x_g, y_g, \varphi_g\rangle$ and use cubic Bézier curves for specifying the trajectories to be followed (figure 2). With $t \in [0, 1]$ a cubic Bézier curve is defined as $B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^(t-1) P_2 + t^3 P_3$.

The P_i are called control points. For the navigation problem P_0 is the starting position and P_3 is the goal position. With the orientation given at P_0, P_1 is determined by the distance to P_0, for P_3 and P_2 respectively. So for us a Bézier curve is determined by the two distances d_0 and d_1.

```
control routine navigation ($\langle x_g, y_g, \varphi_g\rangle$)
  trajectory := $\boxed{\text{calculate-bezier-curve}}$($\langle x, y, \varphi\rangle$, $\langle x_g, y_g, \varphi_g\rangle$)
  do
    trajectory-point := pop(trajectory)
    do
      $\boxed{\text{get-next-command}}$($\langle x, y, \varphi\rangle$, trajectory-point)
    until at-point(trajectory-point)
  until at-point($\langle x_g, y_g, \varphi_g\rangle$)
```

Figure 3: Overview of our navigation routine.

We decompose the navigation problem into two subproblems as shown in figure 3: (1) Finding a suitable Bézier curve. We call this the high-level navigation problem. (2) Navigating from a point on the curve to the next trajectory point. This we refer to as the low-level navigation problem.

To keep the case study simple, we abstract away from other parameters that influence the performance of the navigation routine such as the density of trajectory points on the Bézier curve. They have small impact on performance and we keep them constant in our experiments.

Finding a Trajectory

In our case study we have investigated three different approaches for the implementation of the function calculate-bezier-curve: a fully programmed one that performs an exhaustive search through all possible solutions; a heuristic one, and a learned one.

The fully programmed function produces very good results, but is too slow to be used at execution time. The heuristic solution calculates d_0 and d_1 independently, based on the angle deviation of each point with respect to the line of sight between the two points. A third solution is to learn the function calculate-bezier-curve by experience. Unfortunately, there is a strong dependency between the parameters d_0 and d_1. Therefore we split the problem into two learning tasks: one to determine only d_0, the other one to determine d_1 as a function of d_0.

Low-Level Navigation

In this section, we explain the solution of the low-level navigation problem in more detail. We demonstrate the explicit specification of a learning problem and show different alternatives for solving the navigation task.

The low-level navigation is simpler than the original navigation problem in that the target points are rather close, so that the goal angle can be omitted. The orientation of the overall navigation task is achieved by following the Bézier curve.

(1) Parameterization of the Experience Abstraction. The choice of abstractions determines how concisely situations and tasks can be represented and how strongly the characterizations of situations and tasks correlate with the concepts to be learned. The abstraction has a strong effect on the performance of the learning process.

Two possible feature languages with their respective abstractions are shown in table 1. The first one describes the distance between the start and goal point and the angle of the start point relative to the line of sight. In the second possibility an arc is drawn between the start and goal point with the orientation vector at the start point as a tangent. Here the abstraction is described in terms of the radius r_c of this arc.

(2) Parameterization of the Experience Distribution. It is often useful to specify more than one distribution for obtaining a broader variety of experiences. Therefore we define the relevant parameters for the distribution first.

```
distribution parameters nav distribution
  ⟨x, y, $\varphi$⟩$_{start}$: constant ⟨-5.0, -2.5, 0.0⟩
  $\varphi_{end}$:       constant 90.0
  rotation:      range ⟨1.0, 180.0⟩
  translation:   range ⟨0.0, 1.0⟩
```

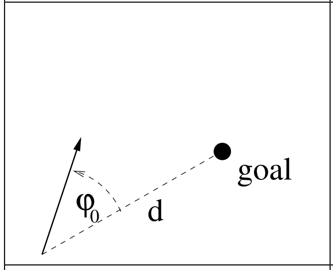
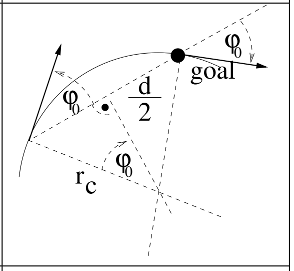
features 1	features 2
	
$\mathsf{d} \leftarrow \sqrt{(\mathsf{x_g} - \mathsf{x})^2 + (\mathsf{y_g} - \mathsf{y})^2}$ $\varphi_0 \leftarrow \left\lvert \varphi - \mathsf{arctan}\left(\frac{\mathsf{x_g} - \mathsf{x}}{\mathsf{y_g} - \mathsf{y}}\right)\right\rvert$	$r_c \leftarrow \frac{\sqrt{(x_g-x)^2+(y_g-y)^2}}{2\sin\varphi_0}$
$\mathsf{d} \times \varphi_0 \rightarrow \mathsf{v} \times \omega$ (AB 1)	$\mathsf{r_c} \rightarrow \mathsf{v} \times \omega$ (AB 2)
abstraction 1	abstraction 2

Table 1: Different parameterizations for feature language and abstraction.

Now different distributions can be defined by setting the values of the non-constant parameters. The values of a parameter can be obtained systematically, randomly or by a list of fixed values. If not stated otherwise, the parameters are assumed to be independent, although distributions over combinations of parameters can be defined as well.

```
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
```

Instead of defining experience distributions, it is possible that the robot use experiences aquired during its operation. In robot soccer, we can use the experiences made during games.

(3) Methods for Recognizing Experiences. To collect experiences the robot has to recognize them and detect failures during their acquisition. In our example an experience starts at a certain point and terminates when the robot has reached a certain turning angle. Furthermore we check if the robot has gone out of the field or exhausted the given time resources.

The methods for gathering experiences are usually straightforward, so one doesn't have to experiment with them the way one does with the other parameters like experience abstraction or distribution.

(4) Parameterization of the Experience Extraction. As described in section , a learning problem is defined by a set of experiences and a learning element. The experiences are extracted from an episode database.

Table 2 shows the specifications of two possible sets of experiences. The first one is trivial and uses all the available experiences. The second one is more sophisticated in that it selects only fast examples. The table match-nav is obtained by applying a matching operator as described in (Galhardas *et al.* 2001) on the stored examples, so that similar routes are grouped together.

```
CREATE MATCHING match-nav
FROM nav-exp ne1, nav-exp ne2
LET distance = pathSimilarity(ne1.id, ne2.id)
WHERE distance < maxDist(ne1.id, ne2.id, δ)
INTO match-exp
```

ES 1	**def-experience-set** all-experiences
	SELECT id
	FROM nav-exp

ES 2	**def-experience-set** fast-experiences
	SELECT DISTINCT time,id
	FROM (SELECT time,id,id1 FROM 'nav-exp' ne
	JOIN 'match-exp' me ON ne.id=me.id2) t1
	JOIN (SELECT MIN(time) mt,id1 FROM 'nav-exp' ne
	JOIN 'match-exp' me ON ne.id=me.id2
	GROUP BY id1) t2
	ON t1.time=t2.mt AND t1.id1=t2.id1;

Table 2: Different parameterizations for experience set.

(5) Parameterization of the learning element. Now having chosen the experiences that are to be used for learning, we only have to describe the parameters of the learning element. One parameterization could be

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

(6) Parameterization of the amount of programming. Often learning alone is not enough to solve complex problems. For instance, our learning approach to the low-level navigation has one fundamental drawback. It is hard to decide whether a navigation routine is better than another. There can be cases when a routine is fast, but inaccurate. Depending on the situation the robot must have access to navigation routines with different qualities. Any learned low level navigation routine can only be optimized under one criterion.

Instead of learning completely different routines, we can reformulate the learning problem by inverting the abstraction:

abstraction nav abstraction

$\mathsf{v} \times \omega \rightarrow \mathsf{r_c}$ (AB 3)

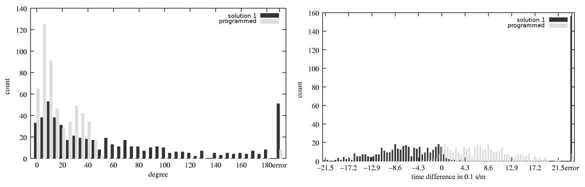
Here we know our translational and rotational velocities and are interested in the arc the robot will go with these parameters. The learned function can now be used for a search algorithm. We optimized the function so that the robot goes as fast as possible while rotating as little as possible. But it would be easy to write functions with different criteria.

Experimental Results

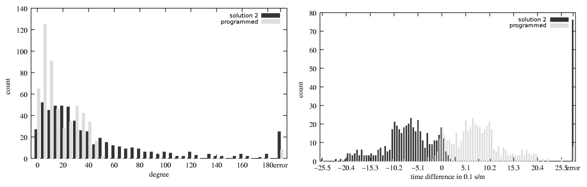
In this section we present experimental results that were obtained by combining the different parameters explained in the previous section. The following table gives an overview of our solutions.

solution	1	2	3	4
abstraction	AB 1	AB 1	AB 1	AB 3
experience set	ES 1	ES 1	ES 2	ES 2
programming	none	none	little	yes
high-level	heuristic	learned	learned	learned

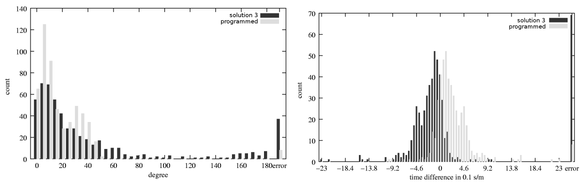
Experiments. In our experiments we gave the robot several navigation tasks where it had to reach a point with a certain orientation. Only runs that reached the point within a given radius were considered successful. The successful



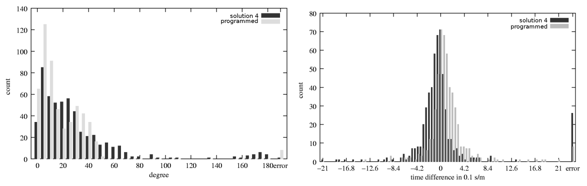
(a) Solution 1



(b) Solution 2



(c) Solution 3



(d) Solution 4

Figure 4: Comparison of different parameterizations. In each case the left diagram shows the accuracy of the goal angle, the right one the time difference compared to a completely programmed routine, which is drawn in light grey.

runs were then categorized along two lines: the accuracy the desired orientation was achieved with and the time needed.

In section we described a fully programmed routine that provides good solutions, but is intractable for real-time applications. Since in our simulation the computation time for finding a solution can be disregarded, we consider this routine as the best possible solution and therefore compare our (partly) learned solutions to this reference routine.

The diagrams for the accuracy in figure 4 give the angle deviation at the goal point. At the rightmost side the cases are denoted when the routine didn't reach the point at all.

In the time diagrams the time difference of two routines is shown. When comparing two identical routines, the diagram shows two bars of equal height at the origin. The faster a routine, the more bars of its color are on the right hand side. In the time comparison, runs are considered successful only when the accuracy is better than $45°$.

Results. In the first trial we used the simple heuristic approach for calculating the Bézier curve. For the low-level navigation we used abstraction AB1 and all experiences without filtering. Figure 4(a) shows the performance of this configuration. This routines does very poorly. It is practically always slower than our reference routine and it often differs from the desired goal angle by more than $90°$.

The second solution only differs from the first in the calculation of the Bézier curve. This time the parameters are learned from experiences. The result of this configuration is shown in figure 4(b). We see a slight improvement in accuracy, although it is still far from satisfactory. Similarly the programmed method still runs faster in almost every case, although the difference is smaller now.

In the third experiment we only used the fastest examples and a little programming was added, so that the robot turns at the beginning of the trajectory when the turning angle towards the goal point was near $180°$. This time the effect is more notable. Now most of the points are reached with an acceptable angle deviation. Furthermore the time statistics have shifted significantly. It is sometimes faster or not much slower than the reference routine.

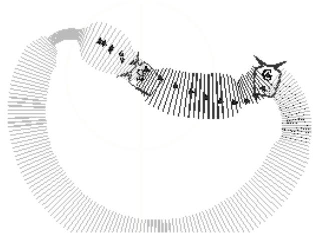


Figure 5: Trajectories of solution 2 (light gray) and 3.

The performance gain can also be seen when comparing the navigated trajectories. Figure 5 shows a navigation task performed by solution 2 and 3. Whereas the former can't follow the desired Bézier curve at all and takes a long way round, the latter approach follows the trajectory perfectly.

Finally we implemented a combined approach of programming and learning as described in section . As shown in figure 4(d), the performance almost reaches the level of the programmed one.

Discussion

Let us now discuss some of the issues of tightly embedding learning into autonomous robot control. (1) Section shows that aspects of autonomous robot learning can be specified in RoLL explicitly and transparently. Section shows that the specification of these aspects has a large impact on the performance of learned routines. In most learning robot controllers these aspects are adressed mainly implicitly or not at all. Turning them into pieces of code of the control program substantially improves the engineering methodolgy for learning robot conbtrollers. (2) The specifications listed in section are complete. Taken together they specify the complete learning process including experience acquisition, feature abstraction, monitoring of experience collection, parameterization of the learning algorithm. The learning problem specifications are therefore completely executable by the RoLL interpreter. (3) RoLL allows for the specification and simultaneous experimentation with different variants of the learning problem. It thereby encourages and simplifies the experimentation and the empirical comparison of different variants. (4) We have also seen the seamless transition between, and a mixing of, learning and programming. Programmed code pieces can be simply substituted by specifications of learning problems that are also executable in the respective context.

In our view, these are critical functionalities that robot control languages must provide in order to allow us programmers to implement learning robot controllers for complex and dynamically changing tasks that can compete in terms of performance with their hand-coded counterparts. We believe that such programming language functionality is necessary to further promote the application of autonomous learning mechanisms in robot control.

Related Work

We are not aware of any work where aspects of learning problems are systematically changed and compared on the scale of our work. Empirical evaluation is an important issue in robotics. CLIP/CLASP (Anderson *et al.* 1995) is a macro extension of LISP, which supports the collection of experimental data and its empirical analysis. Other interesting approaches for the comparison of components in robot control systems are found in the work of Guttman and Fox [1998; 2002]. However, they only treat very selected and restricted aspects of robot control.

We use the language RoLL, because it provides most of the concepts we are interested in. There are several other programming language we have considered for this purpose, but that didn't quite satisfy our requirements. Thrun (Thrun 2000) 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. Programmable Reinforcement Learning Agents (Andre & Russell 2001) 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 better learning by acquiring and selecting the data used for learning.

Conclusion

Proper embedding and parameterization of learning mechanisms is a necessary precondition for successful robot learning. In this paper we have performed a case study that has supported this point. We have used RoLL, an extension of the robot control language RPL that allows for the explicit and transparent specification of learning problems, their embedding into robot control, and the parameterization of the learning mechanisms. Using RoLL we could make aspects of learning explicit that are typically neglected or only implicitly modeled in robot control. The parameters that we have controlled using RoLL include state space transformations, reformulations of the learning problems, filtering experiences, and reasoning about the performance of learning mechanisms. In our experiments we could enhance the learning performance significantly through adequate choice of the parameter settings.

We have also seen that the explicit specification of learning problems has additional benefits. The declarativity of RoLL's control structures has substantially improved the readability of the program and made the solutions to learning problems understandable. Thus learning problems can be carried over to similar problems or other robot platforms with minimal modifications. Parameterizations of learning problems can be compared easily, which leads to a faster development of high quality solutions.

We have further seen that a smooth interlinkage of classical programming and learning algorithms yields solutions that can neither be achieved by learning nor programming alone. With an integration of learning into programming, robot controllers can be developped more quickly and more robustly.

References

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

Andre, D., and Russell, S. 2001. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, 1019–1025. Cambridge, MA: MIT Press.

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

Beetz, M.; Kirsch, A.; and Müller, A. 2004. RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning. In *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*.

Galhardas, H.; Florescu, D.; Shasha, D.; Simon, E.; and Saita, C.-A. 2001. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the 27th VLDB Conference*.

Gutmann, J.-S.; Burgard, W.; Fox, D.; and Konolige, K. 1998. An experimental comparison of localization methods. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Gutmann, J.-S. Fox, D. 2002. An experimental comparison of localization methods continued. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

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

Thrun, S. 2000. 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: IEEE.