Optimized Execution of Action Chains Using Learned Performance Models of Abstract Actions*

Freek Stulp and Michael Beetz
Intelligent Autonomous Systems Group, Technische Universität München
Boltzmannstrasse 3, D-85747 Munich, Germany
{stulp,beetz}@in.tum.de

Abstract

Many plan-based autonomous robot controllers generate chains of abstract actions in order to achieve complex, dynamically changing, and possibly interacting goals. The execution of these action chains often results in robot behavior that shows abrupt transitions between subsequent actions, causing suboptimal performance. The resulting motion patterns are so characteristic for robots that people imitating robotic behavior will do so by making abrupt movements between actions.

In this paper we propose a novel computation model for the execution of abstract action chains. In this computation model a robot first learns situation-specific performance models of abstract actions. It then uses these models to automatically specialize the abstract actions for their execution in a given action chain. This specialization results in refined chains that are optimized for performance. As a side effect this behavior optimization also appears to produce action chains with seamless transitions between actions.

1 Introduction

In recent years, a number of autonomous robots, including WITAS [Doherty *et al.*, 2000], Minerva [Thrun *et al.*, 1999], and Chip [Firby *et al.*, 1996], have shown impressive performance in long term demonstrations. These robots have in common that they generate, maintain, and execute chains of discrete actions to achieve their goals. The use of plans enables these robots to flexibly interleave complex and interacting tasks, exploit opportunities, and optimize their intended course of action.

To allow for plan-based control, the plan generation mechanisms are equipped with libraries of actions and causal models of these actions. These models include specifications of action effects and of the conditions under which the actions are executable. For good reasons, the action models are specified abstractly and disregard many aspects of the situations before, during, and after their execution. This abstractness has several big advantages. Programmers need to supply fewer actions because viewed at an abstract level the actions are applicable to a broader range of situations. Also the models themselves become more concise. This not only eases the job of the programmers but also the computational task of the automatic planning systems. At more abstract levels the search space of plans is substantially smaller and fewer interactions between actions need to be considered.

The advantages of abstraction, however, come at a cost. Because the planning system considers actions as black boxes with performance independent of the prior and subsequent steps, the planning system cannot tailor the actions to the contexts of their execution. This often yields suboptimal behavior with abrupt transitions between actions. The resulting motion patterns are so characteristic for robots that people trying to imitate robotic behavior will do so by making abrupt movements between actions. In contrast, one of the impressive capabilities of animals and humans is their capability to perform chains of actions in optimal ways and with seamless transitions between subsequent actions.

Let us illustrate these points using a simple scenario in autonomous robot soccer that is depicted in Figure 1. The starting situation is shown in the left sub-figure. The planner issues a three step plan: 1) go to the ball; 2) dribble the ball to shooting position; 3) shoot. If the robot naively executed the first action (as depicted in the center sub-figure), it might arrive at the ball with the goal at its back. This is an unfortunate position from which to start dribbling towards the goal. The problem is that in the abstract view of the planner, being at the ball is considered sufficient for dribbling the ball and the dynamical state of the robot arriving at the ball is considered to be irrelevant for the dribbling action.

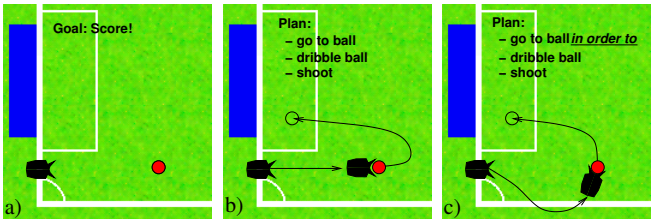


Figure 1: Alternative execution of the same plan

What we would like the robot to do instead is to go to the ball *in order* to dribble it towards the goal afterwards. The

*The work described in this paper was partially funded by the Deutsche Forschungsgemeinschaft in the SPP-1125.

robot should, as depicted in the Figure 1c, perform the first action sub-optimally in order to achieve a much better position for executing the second plan step. The behavior shown in Figure 1c exhibits seamless transitions between plan steps and has higher performance, achieving the ultimate goal in less time.

In this paper we propose a novel computational model for plan execution that enables the planner to keep its abstract action models and that optimizes action chains at execution time. The basic idea of our approach is to learn performance models of abstract actions off-line from observed experience. These performance models are rules that predict the situation- and parameterization-specific performance of abstract actions, e.g. the expected duration. Then at execution time, our system determines the set of parameters that are not set by the plan and therefore define the possible action executions. It then determines for each abstract action the parameterization such that the predicted performance of the action chain is optimal.

The technical contributions of this paper are threefold. First, we propose a novel computational model for the execution time optimization of action chains, presented in section 2. Second, we show how situation-specific performance models for abstract actions can be learned automatically (section 3). Third, we describe a mechanism for subgoal (post-condition) refinement for action chain optimization. We apply our implemented computational model to chains of navigation plans with different objectives and constraints and different task contexts (section 4). We show for typical action chains in robot soccer our computational model achieves substantial and statistically significant performance improvements for action chains generated by robot planners (section 5).

2 System overview

This section introduces the basic concepts upon which we base our computational model of action chain optimization. Using these concepts we define the computational task and sketch the key ideas for its solution.

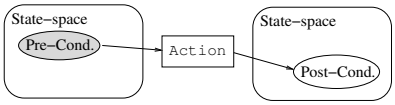
2.1 Conceptualization

Our conceptualization for the computational problem is based on the notion of actions, performance models of actions, teleo-operators, teleo-operator libraries, and chains of teleo-operators. In this section we will introduce these concepts.

Actions are control programs that produce streams of control signals, based on the current estimated state, thereby influencing the state of the world. One of the actions used here is `goToPose`, which navigates the robot from the current pose (at time t) $[x_t,y_t,\phi_t]$ to a future destination pose $[x_d,y_d,\phi_d]$ by setting the translational and rotational velocity of the robot:

`goToPoseAction`$(x_t,y_t,\phi_t,x_d,y_d,\phi_d) \rightarrow v_{tra},v_{rot}$

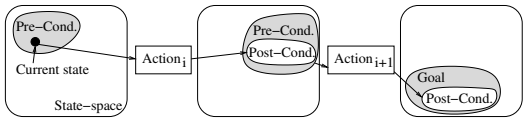
Teleo-operators (TOPs) consist of an action, as well as pre- and post-conditions [Nilsson, 1994]. The post-condition represents the intended effect of the TOP, or its goal. It specifies a region in the state space in which the goal is satisfied. The pre-condition region with respect to a temporally extended action is defined as the set of world states in which continuous execution of the action will eventually satisfy the post-condition. They are similar to Action Schemata or STRIPS operators in the sense that they are temporally extended actions that can be treated by the planner as if they were atomic actions.



The `goToPoseTOP` has the empty pre-condition, as it can be executed from any state in the state space. Its post-condition is $[x_t \approx x_d, y_t \approx y_d, \phi_t \approx \phi_d]$. Its action is `goToPoseAction`.

TOP libraries contain a set of TOPs that are frequently used within a given domain. In many domains, only a small number of control routines suffices to execute most tasks, if they are kept general and abstract, allowing them to be applicable in many situations. Our library contains the TOPs: `goToPoseTOP` and `dribbleBallTOP`.

A **TOP chain** for a given goal is a chain of TOPs such that the pre-condition of the first top is satisfied by the current situation, and the post-condition of each step satisfies the pre-condition of the subsequent TOP. The post-condition of the last TOP must satisfy the goal. It represents a valid plan to achieve the goal.



Performance models of actions map a specific situation onto a performance measure. In this paper the performance measure is time. Alternatives could be chance of success or accuracy. These models can be used to predict the performance outcome of an action if applied in a specific situation, by specifying the current state (satisfying the pre-conditions) and end state (satisfying the post-conditions).

`goToPoseAction.performance`$(x_t,y_t,\phi_t,x_d,y_d,\phi_d) \rightarrow t$

2.2 Computational task and solution idea

The on-line computational task is to optimize the overall performance of a TOP chain. The input consists of a TOP chain that has been generated by a planner, that uses a TOP library as a resource. The output is an intermediate refined subgoal that optimizes the chain, and is inserted in the chain. Executing the TOP chain is simply done by calling the action of each TOP. This flow is displayed in Figure 2.

To optimize action chains, the pre- and post-conditions of the TOPs in the TOP chains are analyzed to determine which variables in the subgoal may be freely tuned. These are the variables that specify future states of the robot, and are not constrained by the pre- and post-conditions of the respective TOP. For the optimization of these free variables, performance models of the actions are required. Off-line, these models are learned from experience for each action in the TOP library. They are used by the subgoal refinement system during execution time, but available as a resource to other systems as well.

One of the big advantages of our approach is that neither TOP library, nor the generation of TOP chains (the planner)

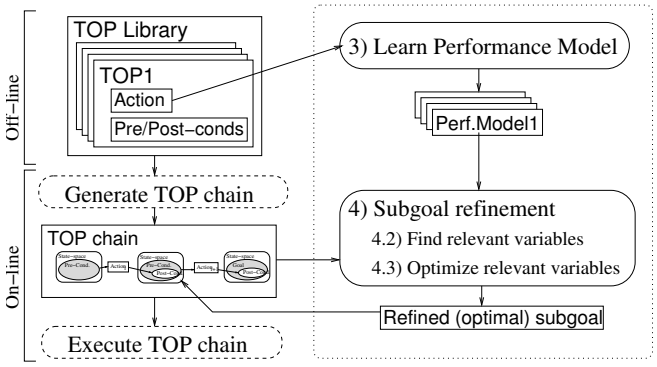


Figure 2: System Overview

nor the TOP chain executor need to be modified in any way to accommodate the action chain optimization system.

3 Learning performance models

The actual optimization of TOP chains, to be discussed in section 4, needs performance models of each action in the TOP library. For each action, the robot learns a function that maps situations to the cost of performing this action in the respective situation. In this paper, the performance measure is time, although our mechanisms applies to other cost functions without requiring any change. The robot will learn the performance function 1) from experience 2) using a transformed state space 3) by partitioning the state space 4) by approximating functions to the data in each of these partitions. We will first motivate *why*, and then explain *how* this has been implemented.

Let us consider the navigation action `goToPoseAction`. This navigation action is based on computing a Bezier curve, and trying to follow it as closely as possible. `dribbleBallAction` uses the same method, but restricts deceleration and rotational velocity, so as not to loose the ball. We abstract away from their implementation, as our methods consider the actions to be black boxes, whose performance we learn from observed experience.

The robot learns the performance function **from experience**. It executes the action under varying situations, observes the performance, and logs the experience examples. Since the method is based solely on observations, it is also possible to acquire models of actions whose internal workings are not accessible. The robot executed each action 1000 times, with random initial and destination poses. The robot recorded the direct variables and the time it took to reach the destination state at 10Hz, thereby gathering 75 000 examples of the format $[x_t,y_t,\phi_t,x_d,y_d,\phi_d,time]$ per action. These examples were gathered using our simulator, which uses learned dynamics models of the Pioneer I platform. It has proven to be accurate enough to port control routines from the simulator to the real robot without change. Using our Pioneer I robots, acquiring this amount of data would take approximately two hours of operation time.

The variables that were recorded do not necessarily correlate well with the performance. We therefore design a **transformed feature space** with less features, but the same potential for learning accurate performance models. In Figure 3 it is shown how exploiting transformational and rotational invariance reduces our original six-dimensional feature space into a three-dimensional one, with the same predictive power.

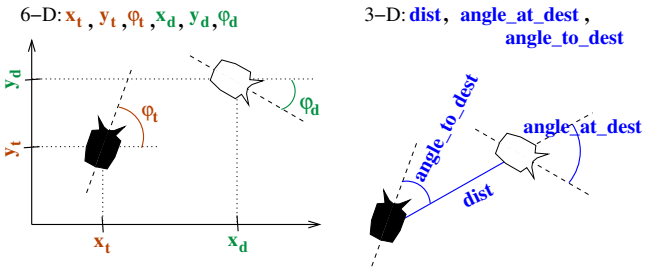


Figure 3: Transformation of the original state space into a lower-dimensional feature space.

Currently, we perform the transformation manually for each action. In our ongoing research we are investigating methods to automate the transformation. By explicitly representing and reasoning about the physical meaning of state variables, we research feature language generation methods.

The last step is to **approximate a function** to the transformed data. This is done using model trees. Model trees are functions that map continuous or nominal features to a continuous value. The function is learned from examples, by a piecewise partitioning of the feature space. A linear function is fitted to the data in each partition. Model trees are a generalization of decision trees, in which the nominal values at the leaf nodes are replaced by line segments. We use model trees because 1) they can be transformed into sets of rules that are suited for human inspection and interpretation 2) comparative research shows they are the best [Belker, 2004; Balac, 2002] 3) They tend to use only relevant variables. This means we can start off with many more features than are needed to predict performance, having the model tree function as an automatic feature selector. The tree was actually learned on an 11-dimensional feature space $[x,y,\phi,x_g,y_g,\phi_g,dx,dy,dist,angle\_to\_dest,angle\_at\_dest]$, the model tree algorithm automatically discovered that only $[dist,angle\_to\_dest,angle\_at\_dest]$ are necessary to accurately predict performance.

We have trained a model tree on the gathered data, yielding rules of which we will now present an example. In Figure 4, we depict an example situation in which $dist$ and $angle\_to\_dest$ are to 2.0m and 0° respectively. Given these values we could plot a performance function for varying values of $angle\_at\_dest$. These plots are also depicted in Figure 4, once in a Cartesian, once in a polar coordinate system. In the linear plot we can clearly see five different line segments. This means that the model tree has partitioned the feature space for $dist$=2 and $angle\_to\_dest$=0 into five areas, each with its own linear model. Below the two plots one of the learned model tree rules that applies to this situation is displayed. An arrow indicates its linear model in the plots.

The polar plot clearly shows the dependency of predicted execution time on the angle of approach for the example situation. Approaching the goal at 0 degrees is fastest, and would take a predicted 2.1s. Approaching the goal at 180

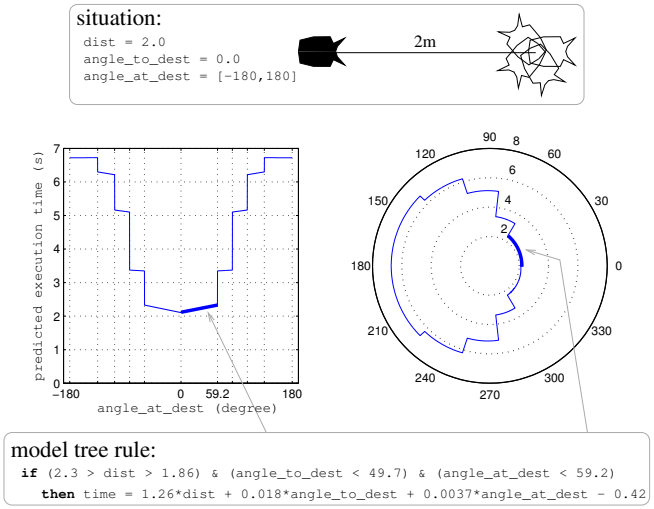


Figure 4: An example situation, two graphs of time prediction for this situation with varying $angle\_at\_dest$, and the model tree rule for one of the line segments.

degrees means the robot would have to navigate around the goal point, taking much longer (6.7s). p To evaluate the accuracy of the performance models, we again randomly generate 1000 new test situations. For the `goToBall` routine, the mean absolute error and root-mean-square error between predicted and actual execution time were 0.31s and 0.75s. For the `dribbleBall` routine these values were 0.29s and 0.73s. As we will see, these errors are accurate enough to optimize action chains.

4 Automatic subgoal refinement

As depicted in Figure 2, the automatic subgoal refinement system takes the performance models and a chain of teleoperators as an input, and returns a refined intermediate goal state that has been optimized with respect to the performance of the overall action chain. To do this we need to specify all the variables in the task, and recognize which of these variables influence the performance and are not fixed. These variables form a search space in which we will optimize the performance using the learned action models.

4.1 State variables

In the dynamic system model [Dean and Wellmann, 1991] the world changes through the interaction of two processes: the *controlling process*, in our case the low-level control programs implementing the action chains generated by the planner, and the *controlled process*, in our case the behavior of the robot. The evolution of the dynamic system is represented by a set of *state variables* that have changing values. The controlling process steers the controlled process by sending *control signals* to it. These control signals directly set some of the state variables and indirectly other ones. The affected state variables are called the *controllable* state variables. The robot for instance can set the translational and rotational velocity directly, causing the robot to move, thereby indirectly influencing future poses of the robot.

For the robot, a subset of the state variables is *observable* to its perceptive system, and they can be estimated using a state estimation module. For any controller there is a distinction between *direct* and *derived* observable state variables. All direct state variables for the navigation task are depicted in Figure 5. Direct state variables are directly provided by state estimation, whereas derived state variables are computed by combinations of direct variables. No extra information is contained in derived variables, but if chosen well, derived variables are better correlated to the control task.

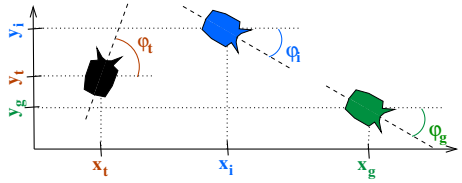


Figure 5: Direct state variables relevant to the navigation task

State variables are also used to specify goals internal to the controller. These variables are *bound*, conform to planning terminology. It is the controller's goal to have the bound internal variables (approximately) coincide with the external observable variables. The robot's goal to arrive at the intermediate position could be represented by the state variables $[x_i,y_i]$. By setting the velocities, the robot can influence its current position $[x_t,y_t]$ to achieve $[x_t \approx x_i, y_t \approx y_i]$.

4.2 Determining the search space

To optimize performance, only variables that actually influence performance should be tuned. In our implementation, this means only those variables that are used in the model tree to partition the state space at the nodes, or used in the linear functions at the leaves.

In both the learned model trees for the actions `goToPoseAction` and `dribbleBallAction`, the relevant variables are $dist$, $angle\_to\_dest$ and $angle\_at\_dest$. These are all derived variables, computed from the direct variables $[x_t,y_t,\phi_t,x_i,y_i,\phi_i]$ and $[x_i,y_i,\phi_i,x_g,y_g,\phi_g]$, for the first and second action respectively. So by changing these direct variables, we would change the indirect variables computed from them, which in effect would change the performance.

But may we change all these variables at will? Not x_t,y_t, or ϕ_t, as we cannot simply change the current state of the world. Also we may not alter bound variables that the robot has committed to, being $[x_i,y_i,x_g,y_g,\phi_g]$. Changing them would make the plan invalid.

This only leaves the free variable ϕ_i, the angle at which the intermediate goal is approached. This acknowledges our intuition from Figure 1 that changing this variable will not make the plan invalid, and that it will also influence the overall performance of the plan. We are left with a one-dimensional search space to optimize performance.

4.3 Optimization

To optimize the action chain, we will have to find those values for the free variables for which the overall performance of the action chain is the highest. The overall performance is

estimated by summing over the performance models of all actions that constitute the action chain. In Figure 6 the first two polar plots represent the performance of the two individual actions for different values of the only free variable, which is the angle of approach. The overall performance is computed by adding those two, and is depicted in the third polar plot.

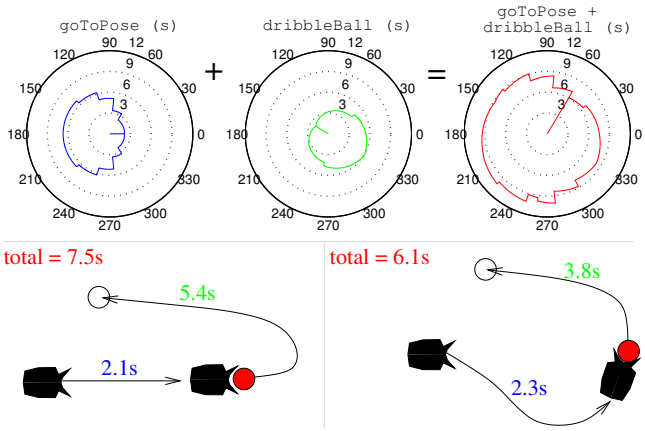


Figure 6: Selecting the optimal subgoal by finding the optimum of the summation of all action models in the chain.

The fastest time in the first polar plot is 2.1s, for angle of approach of 0.0 degrees. The direction is indicated from the center of the plot. However, the total time is 7.5s, because the second action takes 5.4s for this angle . These values can be read directly from the polar plots. However, this value is not the optimum overall performance. The minimum of the overall performance is 6.1s, as can be read from the third polar plot. Below the polar plots, the situation of Figure 1 is repeated, this time with the predicted performance for each action.

We expect that for higher-dimensional search spaces, exhaustive search may be infeasible. Therefore, other optimization techniques will have to be investigated.

5 Results

To determine the influence of subgoal refinement on the overall performance of the action chain, we generated 1000 situations with random robot, ball and final goal positions. The robot executed each navigation task twice, once with subgoal refinement, and once without. The results are summarized in Table 1. First of all, the overall increase in performance over the 1000 runs is 10%. We have split these cases into those in which the subgoal refinement yielded a higher, equal or lower performance in comparison to not using refinement. This shows that the performance improved in 533 cases, and in these cases causes a 21% improvement. In 369 cases, there was no improvement. This is to be expected, as there are many situations in which the three positions are already optimally aligned (e.g. in a straight line), and subgoal refinement will have no effect.

Unfortunately, applying our method causes a decrease of performance in 98 out of 1000 runs. To analyze in which cases subgoal refinement decreases performance, we labeled each of the above runs `Higher`, `Equal` or `Lower`. We then

Before filtering	Total	Higher	Equal	Lower
# runs	1000	533	369	98
improvement	10%	21%	0%	-10%
After filtering	**Total**	**Higher**	**Equal**	**Lower**
# runs	1000	505	485	10
improvement	12%	23%	0%	-6%

Table 1: Results, before and after filtering for cases in which performance loss is predicted.

trained a decision tree to predict this nominal value. This tree yields four simple rules which predict the performance difference correctly in 86% of given cases. The rules declare that performance will stay equal if the three points are more or less aligned, and will only decrease if the final goal position is in the same area as which the robot is, but only if the robot's distance to the intermediate goal is smaller than 1.4m. Essentially, this last rule states that the robot using the Bezier-based `goToBallAction` has difficulty approaching the ball at awkward angles if it is close to it. In these cases, small variations in the initial position lead to large variations in execution time, and learning an accurate, general model of the action fails. The resulting inaccuracy in temporal prediction causes suboptimal optimization. Note that this is a shortcoming of the action itself, not the chain optimization methods. We will investigate if creating a specialized action for the cases in which Bezier based navigation is unsuccessful could solve these problems.

We then gathered another 1000 runs, as described above, but only applied subgoal refinement if the decision tree predicted applying it would yield a higher performance. Although increase in overall performance is not so dramatic (from 10% to 12%), the number of cases in which performance is worsened by applying subgoal refinement has decreased from 98 (10%) to 10 (1%). Apparently, the decision tree correctly filtered out cases in which applying subgoal refinement would decrease performance.

Summarizing: subgoal refinement with filtering yields a 23% increase in performance half of the time. Only once in a hundred times does it cause a small performance loss.

6 Related Work

Most similar to our work is the use of model trees to learn performance models to optimize Hierarchical Transition Network plans [Belker, 2004]. In this work, the models are used to select the next action in the chain, whereas we refine an existing action chain. Therefore, the planner can be selected independently of the optimization process.

Reinforcement Learning (RL) is another method that seeks to optimize performance, specified by a reward function. Recent attempts to combat the curse of dimensionality in RL have turned to principled ways of exploiting temporal abstraction. Several of these *Hierarchical Reinforcement Learning* methods, e.g. (Programmable) Hierarchical Abstract Machines, MAXQ, and Options, are described in the overview paper [Barto and Mahadevan, 2003]. All these approaches use the concept of actions (called 'machines', 'subtasks', or 'options' respectively). In our view, the benefits of our methods are that they acquire more informative per-

formance measures, facilitate the reuse of action models, and scale better to continuous and complex state spaces.

The performance measures we can learn (time, success rate, accuracy) are *informative* values, with a meaning in the physical world. Future research aims at developing meaningful composites of individual models. We will also investigate dynamic objective functions. In some cases, it is better to be fast at the cost of accuracy, and sometimes it is better to be accurate at the cost of speed. By weighting the performance measures time and accuracy accordingly in a composite measure, these preferences can be expressed at execution time. Since the (Q-)Value compiles all performance information in a single non-decomposable numeric value, it cannot be reasoned about in this fashion.

The methods we proposed *scale* better to continuous and complex state spaces. We are not aware of the application of Hierarchical Reinforcement Learning to (accurately simulated) continuous robotic domains.

In Hierarchical Reinforcement Learning, the performance models of actions (Q-Values) are learned in the calling context of the action. Optimization can therefore only be done in the context of the pre-specified hierarchy/program. In contrast, the success of action selection in complex robotic projects such as WITAS, Minerva and Chip depends on the on-line autonomous sequencing of actions through planning. Our methods learn abstract performance models of actions, independent of the context in which they are performed. This makes them *reusable*, and allows for integration in planning systems.

The only approach we know of that explicitly combines planning and RL is RL-TOPS (*Reinforcement Learning - Teleo Operators*) [Ryan and Pendrith, 1998]. Abrupt transitions arise here too, and the author recognizes that "cutting corners" between actions would improve performance, but does not present a solution.

Many behavior based approaches also achieve smooth motion by a weighted mixing of the control signals of various actions [Saffiotti *et al.*, 1995]. Since there are no discrete transitions between actions, they are also not visible in the execution. Since achieving optimal behavior is not an explicit goal, it is left to chance, not objective performance measures.

A very different technique for generating smooth transitions between skills has been developed for quadruped robots [Hoffmann and Düffert, 2004], also in the RoboCup domain. The periodic nature of robot gaits allows their meaningful representation in the frequency domain. Interpolating in this domain yields smooth transitions between walking skills. Since the actions we use are not periodic, these methods do unfortunately not apply.

7 Conclusion and Future Work

On-line optimization of action chains allows the use of planning with abstract actions, without losing performance. Optimizing the action chain is done by refining under-specified intermediate goals, which requires no change in the planner or plan execution mechanisms. To predict the optimal overall performance, performance models of each individual abstract action are learned off-line and from experience, using model trees. It is interesting to see that requiring optimal performance can implicitly yield smooth transitions in robotic and natural domains, even though smoothness in itself is not an explicit goal in either domain.

Applying subgoal refinement to the presented scenario yields good results. However, the computational model underlying the optimization is certainly not specific to this scenario, or to robot navigation. In principle, learning action models from experience using model trees is possible for any action whose relevant state variables can be observed and recorded. The notion of controllable, bound and free state variables are taken directly from the dynamic system model and planning approaches, and apply to any scenario that uses these paradigms. Our future research therefore aims at applying these methods in other domains, for instance robots with articulated arms and grippers, for which we also have a simulator available.

Currently, we are evaluating if subgoal refinement improves plan execution on real Pioneer I robots as much as it does in simulation. Previous research has shown that action models learned in simulation can be applied to real situations with good result [Buck *et al.*, 2002; Belker, 2004].

References

[Balac, 2002] N. Balac. *Learning Planner Knowledge in Complex, Continuous and Noisy Environments.* PhD thesis, Vanderbilt University, 2002.

[Barto and Mahadevan, 2003] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event systems*, 2003.

[Belker, 2004] T. Belker. *Plan Projection, Execution, and Learning for Mobile Robot Control*. PhD thesis, Department of Applied Computer Science, Univ. of Bonn, 2004.

[Buck *et al.*, 2002] S. Buck, M. Beetz, and T. Schmitt. Reliable Multi Robot Coordination Using Minimal Communication and Neural Prediction. In *Selected Contributions of the Dagstuhl Seminar "Plan-based Control of Robotic Agents"*, 2002.

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

[Doherty *et al.*, 2000] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In *Proceedings ECAI-00*, 2000.

[Firby *et al.*, 1996] R. Firby, P. Prokopowicz, M. Swain, R. Kahn, and D. Franklin. Programming CHIP for the IJCAI-95 robot competition. *AI Magazine*, 17(1):71–81, 1996.

[Hoffmann and Düffert, 2004] J. Hoffmann and U. Düffert. Frequency space representation and transitions of quadruped robot gaits. In *Proceedings of the 27th conference on Australasian computer science*, 2004.

[Nilsson, 1994] N.J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1994.

[Ryan and Pendrith, 1998] M. Ryan and M. Pendrith. RL-TOPs: an architecture for modularity and re-use in reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, 1998.

[Saffiotti *et al.*, 1995] A. Saffiotti, K. Konolige, and E.H. Ruspini. A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 1995.

[Thrun *et al.*, 1999] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A tour-guide robot that learns. In *KI - Kunstliche Intelligenz*, pages 14–26, 1999.