Tailoring Action Parameterizations to Their Task Contexts[*]

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

Abstract

Solving complex tasks successfully and efficiently not only depends on *what* you do, but also *how* you do it. Different task contexts have different performance measures, and thus require different ways of executing an action to optimize performance. Simply adding new actions that are tailored to perform well within a specific task context makes planning or action selection programming more difficult, as generality and adaptivity is lost. Rather, existing actions should be parametrized such that they optimize the task-specific performance measure.

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

State-of-the-Art autonomous robot controllers capable of solving a large spectrum of complex tasks are typically equipped with libraries of actions implemented by control routines. The controllers then dynamically combine and partially parameterize these actions on the fly in order to solve the respective set of active tasks.

Consider, for example, the controllers for autonomous soccer robots. These controllers are provided with actions for navigating, kicking, searching, etc. During the game, the controllers dynamically select these actions to perform their immediate tasks. For example, they navigate to the ball in order to get possession of it, or to clear a dangerous situation. In another task context, they navigate in order to dribble the ball towards the opponent's goal. As a consequence, the use of actions in different task contexts require the designer to reason about how the implemented action will perform in these contexts. On the one hand, programmers want to implement the navigation action as fast as possible to be more agile and mobile than the opponents. Unfortunately, fast navigation behavior will cause more frequent and harder collisions with the ball when approaching it and thereby the robot will loose control of the ball. Even worse, while these hard collisions are to be avoided when gaining control of the ball and dribbling, they are often desirable in other task contexts such as clearing a dangerous situation.

Most robot controllers deal with task contexts by providing variants of actions for the different task contexts. A soccer robot programmer provides, instead of a single navigation action, a set of navigation actions such as: `clearBall`, `approachBall`, `dribbleBall`, `interceptBall`, and `blockOpponent`. In the design of the action libraries, most programmers consider a trade-off between the compactness of the action library and its performance. And they are typically willing to sacrifice compactness for performance.

However, having only few abstract actions instead of many specific actions has several advantages. Fewer actions need to be implemented because viewed at an abstract level the actions are applicable to a broader range of situations. At more abstract levels the search space of plans is substantially smaller and fewer interactions between actions need to be considered. This not only eases the job of the programmers but also the computational task of automatic planning systems. Having fewer actions also makes the system more adaptive. Suppose the robots play on a new field on which the dynamics of the robots are very different, and all navigation actions perform badly. If there are many navigation routines, they all have to be retuned, rewritten or relearned to perform well in the new situation. The fewer actions there are, the faster this can be done, and the more adaptive the system is.

In this paper we propose a novel computational model for autonomous robot control that allows the control system to use small sets of general and abstract actions while at the same time achieving the performance of large sets of specialized actions. The computational model performs execution time and context-specific optimization of action plans using learned performance models of the general actions. 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

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

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.

In this paper, we investigate two mechanisms for execution time and context specific action specialization:

1. Specialization of general actions for their improved execution within given action chains.
2. Specialization of actions for predictive failure prevention through subgoal assertion.

The technical contributions of this paper are fourfold.

1. We propose a novel computational model for the execution time optimization and generation of action chains (section 2).
2. We show how situation-specific performance models for abstract actions can be learned automatically, (section 3).
3. 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).
4. We show how performance models can be used to determine when no action can solve the task, and subgoals must be introduced to achieve the goal (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. First of all, we will describe two exemplary scenarios that clarify the problem.

2.1 Two exemplary scenarios

In Figure 1, a typical situation from robotic soccer is shown. The robot's goal is to score a goal. A three step plan suffices to solve this task: 1) go to the ball; 2) dribble the ball to shooting position; 3) kick. If the robot naively executed the first action (as depicted in Figure 1a), 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 or programmer, 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.

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 robot should, as depicted in the Figure 1b, perform the first action sub-optimally in order to achieve a much better position for executing the second plan step. This behavior could be achieved by designing a new action, e.g. `goToPoseInOrderToDribbleTheBallToX`, that takes into account that we plan to dribble the ball to a certain position afterwards. Its long name already indicates the loss of generality, and it is also not guaranteed that this action

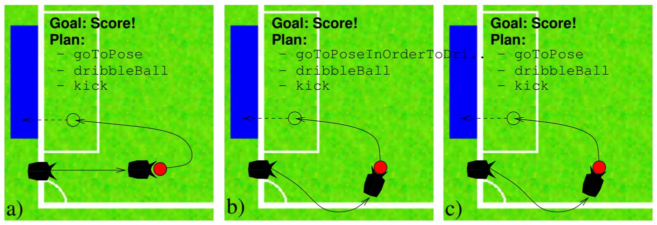


Figure 1: Three alternative plan executions to approach the ball in order to dribble it.

provides the optimal position from which to start dribbling. Preferably, an existing action should be parameterized such that it performs well with respect to the performance measure of the given context. Again, there is also a solution that only uses `goToPose` action. By determining the angle of approach at which the overall performance of the plan is optimal, and parameterizing `goToPose` so that it approaches the ball at this angle, also leads to improved performance. The behavior shown in Figure 1c exhibits seamless transitions between plan steps and has higher performance, achieving the ultimate goal in less time than in Figure 1a. This optimization, called subgoal refinement, can also be automated, as will be demonstrated in section 4.

Another frequent task in robotic soccer is to approach the ball. In Figure 2, the defender's goal is to clear the ball, and it has decided to do so by approaching the ball from behind, and kicking it away from the goal. One way to execute this plan is by first executing its general `goToPose` action. However, since this action does not take the ball into account, it might bump into it before achieving the desired position and orientation, as can be seen in Figure 2a.

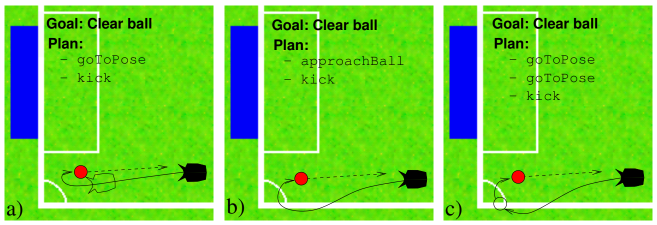


Figure 2: Three alternative plan executions to approach the ball.

To solve this problem, a specialized action that takes the ball into account could be written, e.g. `approachBall`. This variant is shown in Figure 2b, and would work fine. However, there is also a solution that only uses the `goToPose` action, and that does not require us to write `approachBall`. The solution is to introduce an intermediate way-point that ensures there will be no collision with the ball, and performing the navigation task with by appending two `goToPose` actions. Since the chosen path is similar to the path `approachBall` would probably choose, performance is not lost. When a way-point is needed, and where it should lie is determined automatically, using subgoal assertion, which will be presented in section 5.

2.2 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. The basic action we use 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:

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

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.

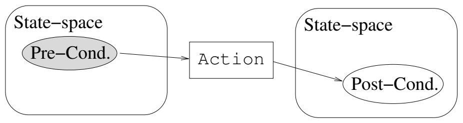


Figure 3: An abstract teleo-operator.

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 `goToPose`.

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.

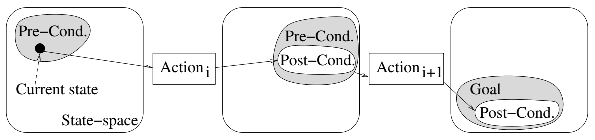


Figure 4: A chain of teleo-operators.

Subgoal refinement is the process of choosing a specific state as a subgoal, from the set of states defined by the post-condition of a preceding and pre-conditions of a subsequent action in a teleo operator chain. In Figure 5, such a specific subgoal has been chosen. This state will be visited in the transition from one action to the next.

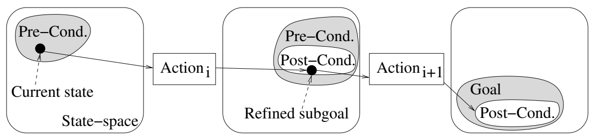


Figure 5: Subgoal refinement.

Performance models of actions map a specific situation onto a performance measure. 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). An example of a performance measure is predicted execution time:

`goToPose.time`$(x_t, y_t, \phi_t, x_d, y_d, \phi_d) \rightarrow t$

2.3 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 6.

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.

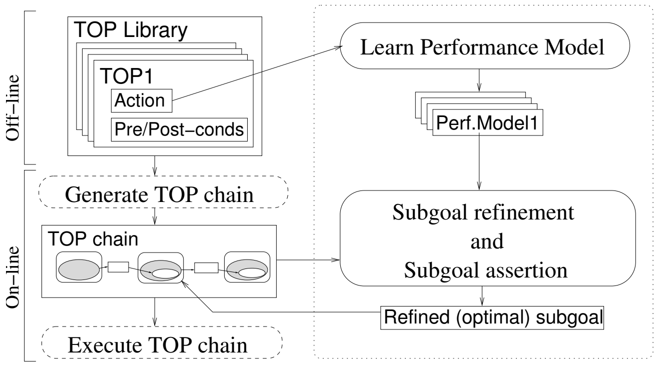


Figure 6: System Overview.

One of the big advantages of our approach is that neither TOP library, nor the generation of TOP chains (the planner) nor the TOP chain executor need to be modified in any way to accommodate the action chain optimization system. We assume that the programmer provides a library of actions containing domain knowledge expressed in the pre- and post-condition, and has mechanisms for generating and executing chains of these actions, be it through planning, arbitration schemes, or simply manual specification. Although each of these components is a research field in its own right, our paper will not focus on them, also to emphasize that our system does not rely on their implementation.

The next three sections describe the main components in Figure 6. In section 3 we describe how performance models of actions are learned from experience. Subgoal refinement and subgoal assertion are presented in sections 4 and 5 respectively.

3 Learning performance models

To perform subgoal refinement and assertion, performance models of each action in the TOP library must be available. For each action, the robot therefore learns a function that maps situations to the cost of performing this action in the respective situation. The robot will approximate the performance function by learning decision and model trees based on observed experience.

Let us consider the navigation action `goToPose`. This navigation action is based on computing a Bezier curve, and trying to follow it as closely as possible [Beetz *et al.*, 2004]. Our `dribbleBall` action 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.

To gather experience, with which the model will be learned, the robot 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 examples are 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.

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 7 it is shown how exploiting transformational and rotational invariance reduces an original six-dimensional feature space into a three-dimensional one, with the same predictive power.

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. Depending on whether a nominal or continuous value needs to be predicted, we use a decision or model tree respectively. Both methods learn a mapping from input features to output feature from experience, by a piecewise recursive partitioning of the examples in feature space. Partitioning continues until all the examples in a partition can be

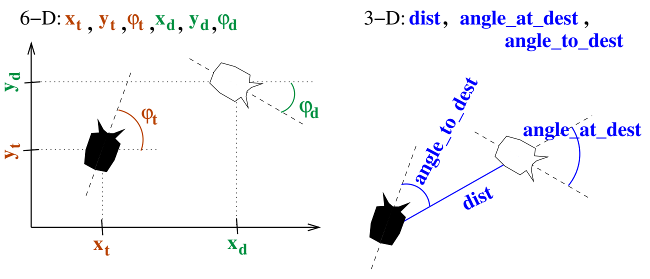

6–D: x_t, y_t, φ_t, x_d, y_d, φ_d
y_d
y_t
φ_t
φ_d
x_t
x_d
3–D: dist, angle_at_dest, angle_to_dest
angle_to_dest
angle_at_dest
dist

Figure 7: Transformation of the original state space into a lower-dimensional feature space.

approximated well by a simple representative model. Decision trees use a nominal value, and model trees a linear function to represent the data in a partition.

We use decision and 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 very appropriate for learning action models [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.

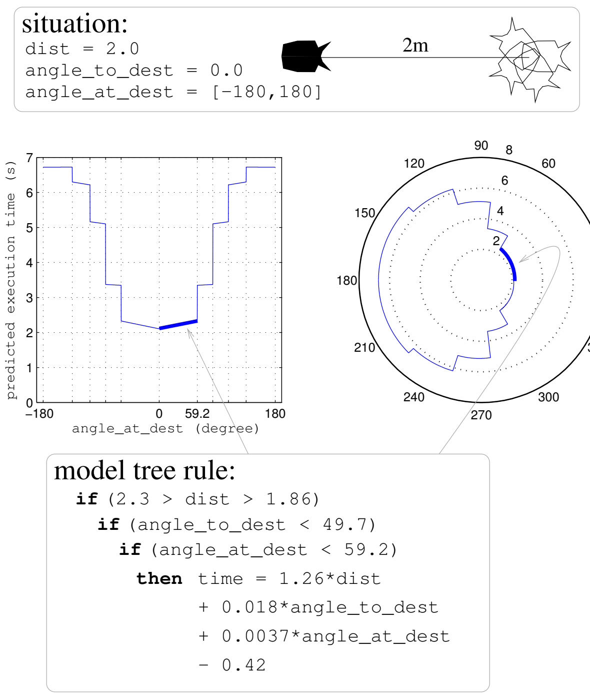
3.1 Prediction of execution duration

The first performance model we have learned is execution duration. It maps a current state and a goal state to the expected time needed to achieve the goal state with this action.

To gather experience, the robot executed each action thousand 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. Using our Pioneer I robots, acquiring this amount of data would take approximately two hours of operation time.

Additional transformed features that were used to learn the model are shown in Figure 7. The model tree was actually learned on an 11-dimensional feature space $[x_t, y_t, \phi_t, x_d, y_d, \phi_d, dx, dy, dist, angle\_to\_dest, angle\_at\_dest]$. The model tree algorithm automatically discovered that only $[dist, angle\_to\_dest, angle\_at\_dest]$ are necessary to accurately predict performance.

We will now give an example of one of the rules learned by the model tree. In Figure 8, 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 8, 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 $dist$=2.0m 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



```
situation:
dist = 2.0
angle_to_dest = 0.0
angle_at_dest = [-180,180]
```

```
model tree rule:
  if (2.3 > dist > 1.86)
    if (angle_to_dest < 49.7)
      if (angle_at_dest < 59.2)
        then time = 1.26*dist
                  + 0.018*angle_to_dest
                  + 0.0037*angle_at_dest
                  - 0.42
```

Figure 8: 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).

To evaluate the accuracy of the performance models, we again randomly executed each action to acquire test examples. For the action `goToPose`, 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.

3.2 Prediction of ball approach failure

The `goToPose` action can often be used well to approach the ball. However, in some situations it will bump into the ball before achieving the desired orientation, as was shown in Figure 2. The second performance model we have learned predicts whether executing `goToPose` will lead to a collision with the ball or not.

To acquire experience, the robot again executed `goToPose` a thousand times, with random initial and destination poses, the ball always positioned at the destination pose. The robot recorded 65 000 training examples of the format $[x_t, y_t, \phi_t, x_d, y_d, \phi_d, collided?]$ per action. The flag *collided?* is set to `Collision` for all the examples in a whole run, if the robot eventually collided with the ball before reaching its desired position and orientation, and to `Success` otherwise.

The model was learned with the same 11-dimensional transformed feature space as used in learning temporal prediction. Again, only $[dist, angle\_to\_dest, angle\_at\_dest]$ were used to predict a collision.

The learned tree, as well as a graphical representation of it, are depicted in Figure 9. The goal pose is represented by the robot, and different areas indicate if the robot can reach this position with `goToPose`, without bumping into the ball first. Remember that `goToPose` has no awareness of the ball at all. The model simply predicts when its execution leads to a collision or not. Intuitively, the rules seem correct. When coming from the right, for instance, it can be seen that the robot always disrespectfully stumbles into the ball, long before reaching the desired orientation. Behind the ball, the robot may not be too close to the ball (checkered area), unless it is facing it. This last rule is indicated by the arrows pointing in the direction of the ball.

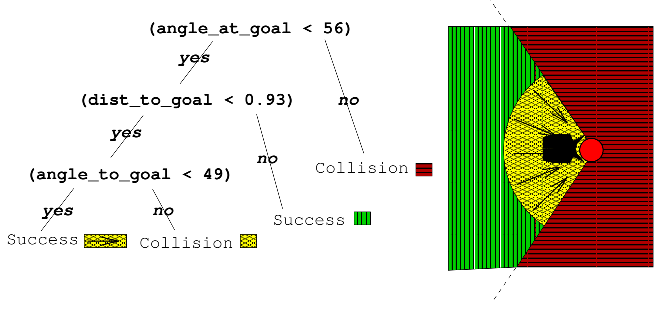


Figure 9: The learned decision tree that predicts whether an unwanted collision will happen.

To evaluate the accuracy of this model, the robot executed another thousand runs, and compared predicted collision with observed collisions. The decision tree predicts collisions correctly in almost 90% of the cases. A more thorough analysis is depicted in Table 1. The model is quite pessimistic, as it predicts failure 61%, whereas in reality it is only 52%. In 10% of cases, it predicts a collision when it actually does not happen. This is preferable to an optimistic model, as it is better to be safe than sorry.

		Observed			Total
		`Coll.`	`Succ.`		Predicted
Predicted	`Coll.`	51%	10%	→	61%
	`Succ.`	1%	38%	→	39%
		↓	↓		↓
Total Observed		52%	48%	→	100%

Table 1: Accuracy of ball collision prediction.

Actually, this decision tree is much more than a performance model. It can be considered as the conditions in which `goToPose` will successfully approach the ball. We now have an teleo-operator `approachBallTOP`, with different preconditions from `goToPoseTOP`. However, since `approachBallTOP` also uses the action `goToPoseTOP` there is no explicit action `approachBall`. We have only determined the conditions under which `goToPose` must be executed to achieve successful ball approach. We will make use of this when applying automatic subgoal assertion in section 5.

4 Automatic subgoal refinement

As depicted in Figure 6, the automatic subgoal refinement system takes the performance models and a chain of teleo-operators 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 10. 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.

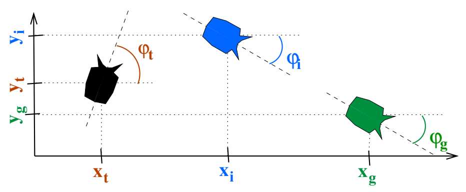


Figure 10: Direct state variables relevant to the navigation task.

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 `goToPose` and `dribbleBall`, 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 11 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.

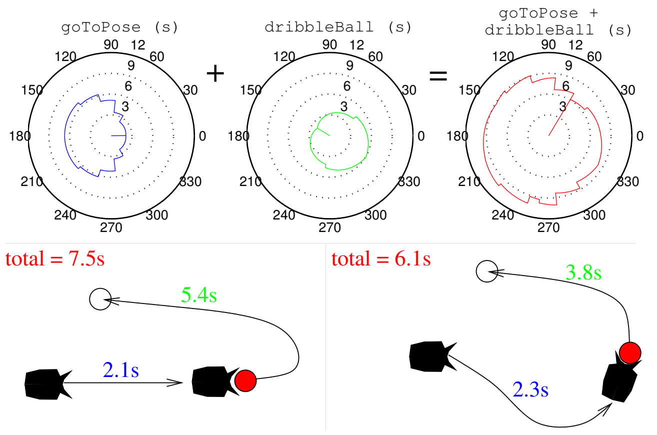


Figure 11: Selecting the optimal subgoal by finding the optimum of the summation of all action models in the chain.

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.

4.4 Results

To determine the influence of subgoal refinement on the overall performance of the action chain, we generated a thousand 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 2. 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.

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

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 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 and a graphical representation are depicted in Figure 12. In this graph, the robot always approaches the centered ball from the left at different distances. The different regions indicate whether the performance increase/decreased due to subgoal refinements, if the goal lies in this region. Three instances with different classification and therefore different colors circles have been inserted.

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 `goToPose` 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.

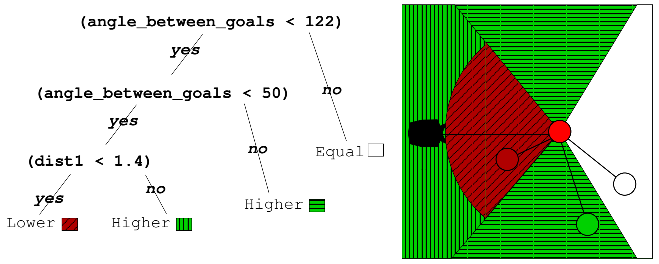


Figure 12: The decision tree that predicts whether subgoal refinement will make the performance better, worse or have no influence at all.

We then performed another thousand test 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.

Without subgoal refinement, the transitions between actions were very abrupt. In general, these 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. 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.

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

5 Automatic subgoal assertion

In the previous section, we have seen how subgoals can be refined in order to optimize performance. In this section, we will show how performance models can be used to detect when the assertion of a new subgoal is necessary.

We use a scenario in which a robot approaches a ball, introduced in section 2.1. A difficulty in approaching the ball is that the robot might collide with the ball before it has reached its desired position and orientation. Since our `goToPose` action is not aware of these potential collisions, it is not always appropriate for approaching the ball. Actually, it can be derived from Table 1 that it fails in 52% of cases. To solve this problem, one could write a new action, e.g. `approachBall`. It would probably be very similar to `goToPose`, but take the ball into account.

Instead of writing a new action, thereby causing the problems discussed in the introduction, it is also possible to reuse `goToPose`, and adapt it to the current context. First of all, it is important to recognize that `goToPose` is actually successful in approaching the ball almost half the time (Table 1). Fortunately, we have models that can predict when success is probable. So when the goal is to approach the ball, and the performance model predicts that `goToPose` can do this collision-free, this action is executed as is.

When no action can be parameterized in such a way that ball approach is likely to succeed, we need to find a chain of actions that can. This is done in a means-ends fashion. First, the robot determines which actions can achieve the goal, and which preconditions must hold for this action to succeed. Then, it determines if any action can achieve these preconditions. In our example, a sequence of two `goToPose` actions can achieve the goal. A constraint is that the second action in the sequence must be able to reach the ball without unintentional collisions. This could be any position in the most left area of Figures 9 and 13, because the performance model predicts that there will be no collision when starting from any of the position in this area.

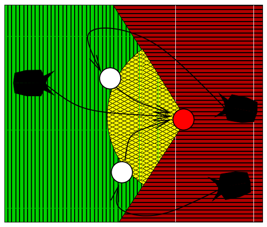


Figure 13: Subgoal assertion to avoid collisions with the ball.

Although all positions in this area can function as an intermediate goal for the two `goToPose` actions, the overall expected execution duration is different for all off them. Therefore, we sample a thousand points from the area, and compute the overall performance by adding the predicted time of the first and second `goToPose` actions in the action chain. The point with the best performance, that is, fastest execution time, is chosen to be the intermediate point. This optimization process is nothing else than subgoal refinement, as has been presented in section 4.

In Figure 13, three instances of the problem are depicted. Since the robot to the left is in the area in which no collision is predicted, it simply executes `goToPose`, without asserting a subgoal. The model predicts that the other two robots will collide with the ball when executing `goToPose`, and a subgoal is asserted. The optimal positions of the subgoals, determined by subgoal refinement, are shown as white circles.

5.1 Results

To evaluate automatic subgoal assertion we executed a thousand random ball approaches, once with assertion, and once without. The results are summarized in Table 3. It is clear that using only `goToPose` is not very successful. It approaches the ball collision-free less than half the time. This is actually exactly what our performance model predicts, as can be seen in Table 1. Applying subgoal assertion dramatically improves this. In less than 3% of cases does the ball approach fail.

We have also investigated under which circumstances a

	direct (no subgoal)	with subgoal assertion
`Success`	47%	97%
`Collision`	53%	3%

Table 3: The effects of applying subgoal assertion to the ball approach task.

subgoal was introduced, and if it was helpful to do so. In 37% of cases, no subgoal was needed, as no collision was predicted. In 52%, a subgoal was asserted, causing a successful completion that was not possible without a subgoal. In 10% of cases, a subgoal was introduced unnecessarily, as the task could have been solved without a subgoal. Note that all these percentages are roughly the same as those in Table 1. Inappropriately introducing the subgoal caused a performance loss of 11% in these cases.

Summarizing: if subgoal assertion is not necessary, it is usually not applied. Half of the time, a subgoal is introduced, which raises successful task completion from 47 to 97%. Infrequently, subgoals are introduced inappropriately, but the performance loss in these cases is an acceptable cost compared to the pay-off of the dramatic increase in the number of successful task completions.

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 [Barto and Mahadevan, 2003]. Several of these *Hierarchical Reinforcement Learning* methods, e.g. (Programmable) Hierarchical Abstract Machines [Parr, 1998; Andre and Russell, 2000], MAXQ [Dietterich, 2000], and Options [Sutton *et al.*, 1999]. 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 performance measures, facilitate the reuse of action models, and scale better to continuous and complex state spaces.

The performance measures we can learn (execution time, action failure) 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 [Doherty *et al.*, 2000], Minerva [Thrun *et al.*, 1999], and Chip [Firby *et al.*, 1996], 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; 1993]. In computer graphics, this approach is called *motion blending*, and is also a wide-spread method to generate natural and fluent transitions between actions, which is essential for lifelike animation of characters. Impressive results can be seen in [Perlin, 1995], and more recently [Shapiro *et al.*, 2003; Kovar and Gleicher, 2003]. Since there are no discrete transitions between actions, they are also not visible in the execution. In all these blending approaches, 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.

Reusing actions and transferring knowledge between them are also key concepts in life-long learning [Thrun and Mitchell, 1993]. This approach exploits the notion that learning to run is much more easy when you already know how to walk, just as approaching a ball is more easy if you already know how to navigate. In [Thrun and Mitchell, 1993], knowledge is transfered between tasks by reusing neural networks that have been trained on one task as a bias for similar, perhaps more complex tasks that have yet to be learned.

7 Conclusion and Future Work

The central idea of this work is that by adapting action parameterization, actions can be tailored to the task context. There is no longer a need to write a new action for each new context, and generality is maintained. Instead of using manual parameterization, we use learned action models to optimize the parameters with respect to the given performance measure.

On-line optimization of action chains allows the use of planning with abstract actions, without losing performance. Optimizing the action chain is done by asserting and 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.

Applying subgoal refinement and assertion to the presented scenarios yields significant performance improvement. 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

[Andre and Russell, 2000] David Andre and Stuart Russell. Programmable reinforcement learning agents. In *Conference on Neural Information Processing Systems (NIPS)*, 2000.

[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.

[Beetz *et al.*, 2004] Michael Beetz, Alexandra Kirsch, and Armin Müller. 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)*, 2004.

[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] Sebastian Buck, Michael Beetz, and Thorsten Schmitt. Reliable Multi Robot Coordination Using Minimal Communication and Neural Prediction. In M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack, editors, *Advances in Plan-based Control of Autonomous Robots. Selected Contributions of the Dagstuhl Seminar "Plan-based Control of Robotic Agents"*, Lecture Notes in Artificial Intelligence. Springer, 2002.

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

[Dietterich, 2000] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[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.

[Kovar and Gleicher, 2003] L. Kovar and M. Gleicher. Flexible automatic motion blending with registration curves. In *Proceedings of ACM SIGGRAPH*, 2003.

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

[Parr, 1998] Ronald Parr. *Hierarchical Control and learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley, 1998.

[Perlin, 1995] Ken Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, 1995.

[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.*, 1993] A. Saffiotti, E. H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Proc. of the IEEE Int. Conf. on Fuzzy Systems*, pages 134–139, San Francisco, California, 1993. IEEE Press.

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

[Shapiro *et al.*, 2003] Ari Shapiro, Frederic Pighin, , and Petros Faloutsos. Hybrid control for interactive character animation. In *Pacific Graphics*, pages 455–461, 2003.

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[Thrun and Mitchell, 1993] S. Thrun and T. Mitchell. Lifelong robot learning. Technical Report IAI-TR-93-7, University of Bonn, Department of Computer Science, 1993.

[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.