Journal of Artificial Intelligence Research 32 (2008) 487-523 Submitted 11/07; published 06/08

Refining the Execution of Abstract Actions with Learned Action Models

Freek Stulp STULP@CS.TUM.EDU
Michael Beetz BEETZ@CS.TUM.EDU
Intelligent Autonomous Systems Group
Technische Universität München
Boltzmannstraße 3, D-85747 Garching bei München, Germany

Abstract

Robots reason about abstract actions, such as *go to position 'l'*, in order to decide what to do or to generate plans for their intended course of action. The use of abstract actions enables robots to employ small action libraries, which reduces the search space for decision making. When executing the actions, however, the robot must tailor the abstract actions to the specific task and situation context at hand.

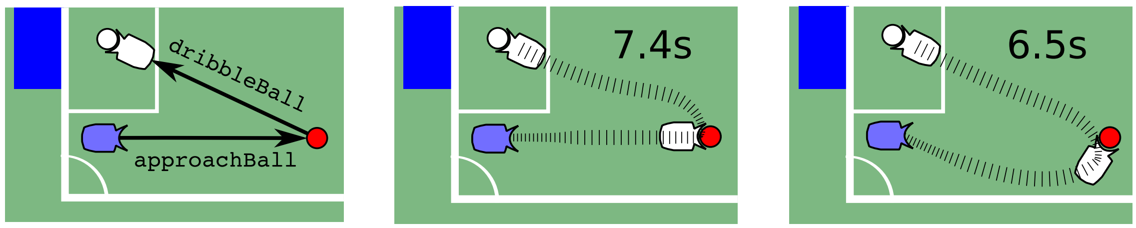
In this article we propose a novel robot action execution system that learns success and performance models for possible specializations of abstract actions. At execution time, the robot uses these models to optimize the execution of abstract actions to the respective task contexts. The robot can so use abstract actions for efficient reasoning, without compromising the performance of action execution. We show the impact of our action execution model in three robotic domains and on two kinds of action execution problems: (1) the instantiation of free action parameters to optimize the expected performance of action sequences; (2) the automatic introduction of additional subgoals to make action sequences more reliable.

1. Introduction

In motor control, the main challenge is to map high-level goals and plans to low-level motor commands. For instance, in the soccer scenario in Figure 1(a), what needs to be done at an abstract level can be informally declared as: "To achieve a scoring opportunity, first approach the ball, and then dribble it towards the opponent's goal." To actually execute this abstract plan, the robot must map it to low-level motor commands, such as translational and rotational velocities.

Using durative actions to bridge this gap has proven to be a successful approach in both nature (Wolpert & Ghahramani, 2000) and robotics (Arkin, 1998). Durative parameterizable actions, or simply *actions*[1], encapsulate knowledge about how certain goals can be achieved in certain task contexts, and produce streams of motor command to achieve these goals. For instance, human and robot soccer players will typically have dribbling, kicking, and passing actions, that are only relevant in the context of soccer. Also, each of these actions achieves different goals within different soccer contexts. The abstract plan is executed by mapping it to a sequence of actions. In the running example, the plan is mapped to the action sequence `approachBall`, `dribbleBall`.

1. In cognitive science, actions are known as *inverse models* or *control rules*, and in robotics also as *behaviors*, *routines*, *policies*, or *controllers*.



(a) Abstract action chain.

(b) Suboptimal execution with an abrupt transition.

(c) Time-optimal execution that exhibits smooth motion.

Figure 1: An abstract action chain, and two alternative executions of this chain. The center of each line on the trajectory represents the robot's position. The lines are drawn perpendicular to the robot's orientation, and their width represents the translational velocity at that point. These values are recorded at 10Hz.

Although these actions specify how to achieve the goal, there are often several ways to execute them. Figure 1 depicts two executions of the same action sequence. In the first, the robot naively executes the first action, and arrives at the ball with the goal at its back, as depicted in Figure 1(b). This is an unfortunate position from which to start dribbling towards the goal. An abrupt transition occurs between the actions, as the robot needs to brake, and carefully maneuver itself behind the ball in the direction of the goal. Preferably, the robot should go to the ball *in order* to dribble it towards the goal afterwards. The robot should, as depicted in Figure 1(c), perform the first action sub-optimally in order to achieve a much better position for executing the second action. In this article, the performance measure is execution duration. The behavior shown in Figure 1(c) is defined to be optimal, as it minimizes the time needed to execute the action sequence.

The reason why such suboptimal behavior is often witnessed in robots is that robot planners, but also designers of robot controllers, view actions at a level of abstraction that ignores the subtle differences between actions. Abstracting away from action details is essential to keep action selection programming and planning tractable. However, 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 execution contexts. This often yields suboptimal behavior with abrupt transitions between actions, as in Figure 1(b). In this example, 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. Whereas the angle of approach is not relevant to the validity of the plan and therefore free to choose, it must be reasoned about in order to optimize plan execution.

So, given an abstract plan, what is the concrete action sequence that is predicted to have minimal total cost (execution duration) and most likely succeed? This problem can be broken down into three subproblems: 1) How can cost and success be predicted? 2) How can action sequences be optimized to have minimal cost? 3) How can sequences of actions be transformed to increase successful execution?

We take inspiration from findings in cognitive science (Wolpert & Flanagan, 2001), and solve the first subproblem by acquiring and applying a third kind of motor control knowledge: being able to predict the outcome and performance of actions. In the running example, if the robot could predict the performance of alternative executions beforehand, it could choose and commit to the fastest execution. To predict the execution duration of action sequences, the robot must predict the execution duration of individual actions. The robot can learn these *action models* through experimentation, observation and generalization. It does so by observing executions of actions with various parameterizations, and learning general models from them with tree-based induction.

We have implemented our approach on a diversity of real and simulated robotic platforms, which will be introduced in the next section. Afterwards, we discuss related work in Section 3. The following four sections are organized according to the flowchart in Figure 2. The representation of (abstract) actions and action models are described in Section 4, along with the generation of abstract plans. We explain how action models are learned from observed experience in Section 5. The first applications of action models, *subgoal refinement*, is described in Section 6. In Section 7, we present *subgoal assertion*, which transforms plans that are predicted to fail into successful plans. Throughout the article, we will describe the representations and algorithms used to implement the flowchart in Figure 2. Finally, we conclude with a summary and outlook on future work in Section 8.

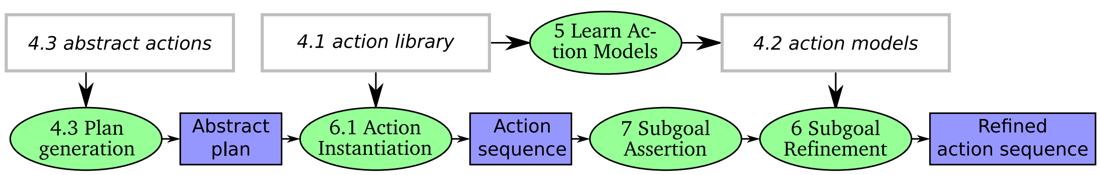


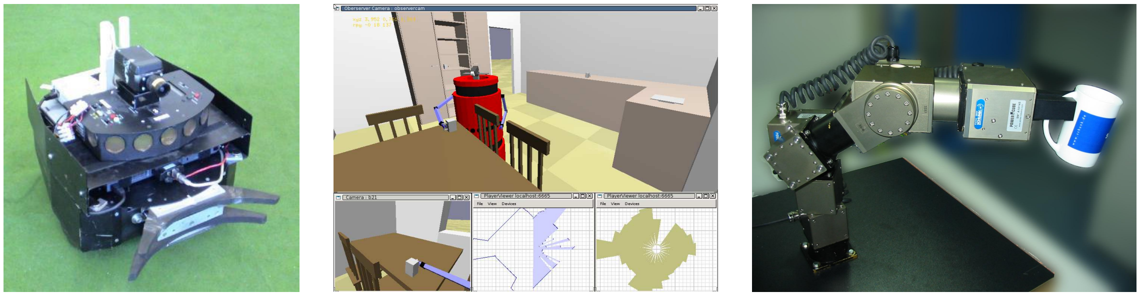
Figure 2: Outline of the article as a flowchart of the overall system.

2. Robotic Domains

In this article, robots learn and apply action models in three domains: robotic soccer, service robotics and arm control. Such a variety of robots and domains has been chosen to emphasize the generality of the approach. Also, the different characteristics of the domains allow different aspects of action model applications to be investigated.

The first domain is robotic soccer (Kitano, Asada, Kuniyoshi, Noda, & Osawa, 1997). In this adversarial domain, performance and efficiency are essential to achieving the goals of the team. Tailoring actions to perform well within a given task context is therefore a necessity. We use the customized Pioneer I robots of our mid-size league team the 'Agilo RoboCuppers' (Beetz, Schmitt, Hanek, Buck, Stulp, Schröter, & Radig, 2004), one of which is depicted in Figure 3(a). These robots have differential drive, and localize themselves with a single forward facing camera. Experiments in this domain were conducted on the real robots, as well as in simulation (Buck, Beetz, & Schmitt, 2002a).

To test our approach on robots with more degrees of freedom, operating in richer environments, we have included an articulated B21 robot in a simulated kitchen environ-



(a) Agilo soccer robot. (b) B21 in the kitchen environment. (c) PowerCube arm.

Figure 3: The three robotic domains considered in this article.

ment (Müller & Beetz, 2006). Household tasks are also less reactive than robotic soccer, and require more complex and long-term planning, which is relevant in the context of the action sequence optimization presented in Section 6. The simulator is based on the Gazebo simulator of the Player Project (Gerkey, Vaughan, & Howard, 2003). The environment, depicted in Figure 3(b) is a typical kitchen scenario, with furniture, appliances, cups and cutlery.

The scenarios in this domain are related to fetching and delivering cups from one place to another. To do so, the robot has navigation actions, and actions for reaching for the cup, or putting it down.

Finally, we evaluate our approach on an articulated manipulator, which performs reaching movements. The experiments were performed with a PowerCube arm from Amtec Robotics with 6 degrees of freedom, shown in Figure 3(c). For the experiments, only two joints were used.

3. Related Work

In this section, we discuss work related to learning predictive models of actions, optimal and hybrid control, as well as learning preconditions.

3.1 Learning Predictive Models of Actions

Jordan and Rumelhart (1992) introduced Distal Learning, which explicitly learns forward models with neural networks. These action models are used to bridge the gap between distal target values and motor commands, which enables motor control learning. Recently, robotic forward models have also been learned using Bayesian networks (Dearden & Demiris, 2005). Infantes, Ingrand, and Ghallab (2006) present recent work that also includes the use of dynamic Bayesian networks. This work extends the approach by Fox, Ghallab, Infantes, and Long (2006), in which action models are learned with Hidden Markov Models. In general, the advantage of Bayesian networks is that they allow the causal nature of a robot's control system to be modelled using a probabilistic framework. However, they are more appropriate for predicting the outcome of a motor command in the next time step, rather than the expected performance of a durative action.

The Modular Selection And Identification for Control (MOSAIC) architecture (Haruno, Wolpert, & Kawato, 2001) also integrates forward models into a computational model for motor control. This framework is intended to model two problems that humans must solve: how to learn inverse models for tasks, and how to select the appropriate inverse model, given a certain task. MOSAIC uses multiple pairs of forward and inverse models to do so. This architecture has not been designed for robot control. We are not aware of (robotic) controllers in which action models are an integral and central part of the computational model, and which are acquired automatically, represented explicitly, and used as modular resources for different kinds of control problems.

Haigh (1998) and Belker (2004) also developed robots that optimize their high-level plans with action models. These models are used to determine the best path in a hallway environment. Our approach rather focuses on optimizing the parameterization of subgoals at fixed points in the plan, and is not limited to navigation plans or tasks. As the authors report good experiences with tree-based induction for learning robot action models from observed data, we have also chosen this learning algorithm.

3.2 Optimal Control

Optimal control refers to the use of online, optimal trajectory generation as a part of the feedback stabilization of a typically nonlinear system (Åström & Murray, 2008). Receding horizon control is a subclass of optimal control approaches, in which an (optimal) trajectory is planned up to a state $x(t + H_p)$ which lies between between the current state $x(t)$ and goal state x_{goal} (Åström & Murray, 2008). After planning the next H_p steps, H_e steps of this trajectory are executed (with $1 \leq H_e \ll H_p$), and a new trajectory is computed from the new current state $x(t + H_e)$ to $x(t + H_e + H_p)$. An example of RHC is the work of Bellingham, Richards, and How (2002), who apply RHC to simulated autonomous aerial vehicles. The rationale behind receding horizon control (RHC) is that there is a diminishing return in optimizing later parts of the trajectory before beginning execution. The experiments described in Section 6.3.1 verify this effect. A comparison between RHC and our methods will also follow in Section 6.3.1.

Todorov, Li, and Pan (2005) take a hierarchical approach to optimal control. A high-level controller uses a more abstract state and command representation, to control a low-level controller, which in its turn controls the physical plant. The main difference with our work is that the purpose of the low-level controller is not to solve a specific subtask, as do our actions, but rather to perform an instantaneous feedback transformation as an abstraction for more high-level controllers.

Using redundant degrees of freedom to optimize subordinate criteria has been well studied in the context of arm control, both in humans (Schaal & Schweighofer, 2005; Wolpert & Ghahramani, 2000; Uno, Wolpert, Kawato, & Suzuki, 1989) and robots (Simmons & Demiris, 2004; Nakanishi, Cory, Mistry, Peters, & Schaal, 2005; Bertram, Kuffner, Dillmann, & Asfour, 2006). Arm poses are said to be redundant if there are many arm configurations that result in an equivalent pose. In the work cited above, all these configurations are called uncontrolled manifold, motion space, or null space, and finding the best configuration is called redundancy resolution or null-space optimization. In the approaches above, optimization is performed analytically, and specific to the arm control domain. Learning the models from observed experience enables whole range of different robots to apply them to a variety tasks.

Smooth motion also arises if the control signals of several actions are combined using a weighted interpolation scheme. Such *motion blending* approaches are found in robotics (Utz, Kraetzschmar, Mayer, & Palm, 2005; Jaeger & Christaller, 1998; Saffiotti, Ruspini, & Konolige, 1993), as well as computer graphics (Perlin, 1995; Kovar & Gleicher, 2003). Since there are no discrete transitions between actions, they are not visible in the execution. Hoffmann and Düffert (2004) propose another method to generate smooth transitions between different gaits of quadruped robots, based on smoothing signals in the frequency domain. Since the actions we use are not periodic, these methods do not apply. In all these approaches, it is assumed that smooth motion will lead to a better performance. However, achieving optimal behavior is not an explicit goal, and no objective performance measures are optimized.

3.3 Reinforcement Learning

Reinforcement Learning (RL) is another method that seeks to optimize performance, specified by a reward function. RL approaches most often model the optimal control problem as a Markov Decision Process (MDP), which is usually solved with dynamic programming. Recent attempts to combat the curse of dimensionality in RL have turned to principled ways of exploiting temporal abstraction (Barto & Mahadevan, 2003). In a sense, Hierarchical Reinforcement Learning algorithms also learn action models, as the Q-value predicts the future reward of executing an action. Note that Q-values are learned for one specific environment, with one specific reward function. The values are learned for all states, but for a single goal function. Our action models are more general, as they describe the action independent of the environment, or the context in which they are called. Therefore, action models can be transfered to other task contexts. Haigh (1998) draws the same conclusion when comparing action models with RL. In our view, our action models also provide more informative performance measures with a physical meaning, such as execution time, and scale better to continuous and complex state spaces.

The only approach we know of that explicitly combines planning and Reinforcement Learning is RL-TOPS (*Reinforcement Learning - Teleo Operators*) (Ryan & Pendrith, 1998). In this approach, sequences of actions are first generated based on their preconditions and effects, using Prolog. Reinforcement Learning within this action sequence is done with HSMQ (Dietterich, 2000). Between actions, abrupt transitions arise too, and the author recognizes that ‘cutting corners’ would improve performance, but does not present a solution.

Most similar to our work, from the point of view of smoothness as an emergent property of optimality requirements with redundant subgoals, is the approach of Kollar and Roy (2006). In this work, a simulated robot maps its environment with range measurements by traversing a set of waypoints. A policy that minimizes the error in the resulting map is learned with RL. As a side-effect, smooth transitions at waypoints arise. This approach has not been tested on real robots.

3.4 Hybrid Control

Problems involving choice of actions and action chains are often regarded as planning problems. However, most planning systems do not aim at optimizing resources, such as time. While scheduling systems are better at representing time constraints and resources, most could not deal with the selection of actions in this problem. Systems that integrate planning and scheduling, such as the work by Smith, Frank, and Jónsson (2000), are able to optimize resources, but ignore interactions between actions and intermediate dynamical states. That is why they do not apply well to continuous domain problems.

In PDDL (Fox & Long, 2003), resource consumption of actions can be represented at an abstract level. Planners can take these resources into account when generating plans. In contrast to such planners, our system generates action sequences that are optimized with respect to very realistic, non-linear, continuous performance models, which are grounded in the real world as they are learned from observed experience.

Other recent approaches to using symbolic planning for robots focus on different problems that arise when such plans are executed on real robots. For instance, Bouguerra and Karlsson (2005) use probabilistic planners in the abstract planning domain, which enables them to exploit uncertainties determined in probabilistic state estimation. aSyMov reasons about geometric preconditions and consequences of actions in a simulated 3-D world (Cambon, Gravot, & Alami, 2004). Note that our methods are complementary rather than incompatible those described by Bouguerra and Karlsson (2005) and Cambon et al. (2004), and merging them would combine their advantages. Cambon et al. (2004) mention that the resulting plan is improved and optimized in some way, but does not describe how. In probabilistic motion planning, such a post-processing step for smoothing the generated paths is a common procedure. Subgoal refinements might well be integrated in this optimization step.

Belker (2004) uses action models learned with model trees to optimize Hierarchical Transition Network (HTN) plans. HTN plans are structured hierarchically from high level goals to the most low level commands. To optimize performance, the order of actions, or the actions themselves are changed at varying levels of the hierarchy. Rather than refining plans, the system modifies the HTN plans themselves, and therefore applies to HTN plans only. On the other hand, we refine an existing action chain, so the planner can be selected independently of the optimization process

Generating collision-free paths from an initial to a final robot configuration is also known as robot motion planning. A common distinction between algorithms that generate such paths is between global, local and hybrid approaches (Brock & Khatib, 1999). The approach presented in this article is a global approach, as the planning is performed offline. One disadvantage of offline algorithms is that there is no guarantee that the planned trajectory will actually succeed on execution. With respect to robotic motion planning, movable obstacles or unforeseen dynamics are just two examples why correctly planned trajectories can fail. The same holds for all global approaches, of which the work by Bouguerra and Karlsson (2005), Cambon et al. (2004) are examples discussed previously. Hybrid motion planning approaches, compute and commit plans offline, but leave freedom in the plans to react to unforeseen circumstances (Brock & Khatib, 1999). Whereas in hybrid motion planning approaches the freedom *between* the subgoals considered, subgoal refinement considers the freedom *at the subgoal itself.* Note that this implies that hybrid approaches are not incompatible with subgoal refinement, and they might complement each other well.

3.5 Learning Preconditions

In Section 7, we demonstrate how failure action models, which predict whether an action will fail or succeed, are learned. This is similar to learning preconditions of actions. In most of the research on learning preconditions, the concept that is being induced is symbolic. Furthermore, the examples consist only of symbols that are not grounded in the real world. The precondition is then learned from these examples, for instance through Inductive Logic Programming (Benson, 1995) or more specialized methods of logic inference (Shahaf & Amir, 2006; Clement, Durfee, & Barrett, 2007). These approaches have not been applied to robots. We believe this is because symbolic representations only do not suffice to encapsulate the complex conditions that arise from robot dynamics and action parameterizations. Such models are best learned from experience, as described in Section 5.

Schmill, Oates, and Cohen (2000) present a system in which non-symbolic planning operators are learned from actual interaction of the robot with the real world. The experiences of the robot are first partitioned into qualitatively different outcome classes, through a clustering approach. The learned operators are very similar to previously hand-coded operators. Once these classes are known, the robot learns to map sensory features to these classes with a decision tree, similar to our approach. This approach aims at learning to predict what the robot will perceive after executing an action from scratch, whereas condition refinement aims at refining an already existing symbolic preconditions based on changing goals.

Buck and Riedmiller (2000) propose a method for learning the success rate of passing actions in the context of RoboCup simulation league. Here a neural network is trained with 8000 examples in which a pass is attempted, along with the success or failure of the pass. This information is used to manually code action selection rules such as 'Only attempt a pass if it is expected to succeed with probability > 0.7'. This is also a good example of integrating human-specified and learned knowledge in a controller.

Sussman (1973) was the first to realize that *bugs* in plans do not just lead to failure, but are actually an opportunity to construct improved and more robust plans. Although this research was done in the highly abstract symbolic blocks world domain, this idea is still fundamental to transformational planning. In the XFRMLearn framework proposed by Beetz and Belker (2000), human-specified declarative knowledge is combined with robot-learned knowledge. Navigation plans are optimized with respect to execution time by analyzing, transforming and testing structured reactive controllers (SRCs) (Beetz, 2001). One difference with XFRMLearn is that in our work, the analysis phase is learned instead of human-specified. Another difference is that XFRMLearn improves existing plans, whereas condition refinement learns how to adapt to changing action requirements, such as refined goals.

4. Conceptualization

Our conceptualization for the computational problem is based on the dynamic system model (Dean & Wellmann, 1991). In this model, the world changes through the interaction of two processes: the *controlled process* and the *controlling process*. The controlled process is essentially the *environment*, including the body and *sensors* of the robot. In the dynamic system model, it is assumed that the environment can be described by a set of *state variables*.

The controlling process performs two tasks. First, it uses *state estimation* to estimate the values of the state variables in the environment, based on the *percepts* that the sensors provide. The observable state variables are encapsulated in the *belief state*. Second, the *controller* takes a belief state as input, and determines appropriate *motor commands* that direct the state variables to certain target values. These motor commands are dispatched to the motor system of the robot in the environment. For each robotic domain, the state estimation methods, the state variables in the belief state, and motor commands are listed in Table 5 in Appendix A. The rest of this section will focus on the concepts used inside the controller.

4.1 Actions and Action Libraries

Actions are control programs that produce streams of motor commands, based on the current belief state. In robotics, actions usually take parameters that allow them to be used in a wide range of situations. Instead of programming an action `dribbleBallToCenter`, it is preferable to program an action `dribbleBall(Pose)` that can dribble the ball to any location on the field, including the center. Because actions only apply to certain task contexts, they are easier to design or learn than a controller that must be able to deal with all possible contexts (Haruno, Wolpert, & Kawato, 1999; Jacobs & Jordan, 1993; Ginsberg, 1989).

One of the actions used is `goToPose`, which navigates the robot from the current state $[x,y,\phi,v]$ to a future target state $[x_g,y_g,\phi_g,v_g]$ by setting the translational and rotational velocity $[v,\dot{\phi}]$ of the robot. Table 6 in Appendix A lists the actions used by the different robots. The action parameters are separated into the values of variables in the current state and target values for those variables. The first are the so-called observable state variables, whereas the second exist only internally in the controller. As action models are learned from observed experience, learning and applying them is independent of action implementations. We do not describe the hand-coded implementations here, but rather refer to Stulp (2007, Appendix A).

Action libraries contain a set of actions that are frequently used within a given domain. If each action is designed to cover a large set of tasks, usually only a small set of actions is needed to achieve most tasks in a given domain. Having only a few actions has several advantages: 1) The controller is less complex, making it more robust. 2) Fewer interactions between actions need to be considered, facilitating action selection design and autonomous planning. 3) If the environment changes, only a few actions need to be redesigned or relearned, making the system more adaptive, and easier to maintain.

In this article, we assume that actions are 'innate', and do not change over time. They are the building blocks that are combined and concatenated to solve tasks that single action could not achieve alone. Actions *themselves* are never modified or optimized in any way. Rather, their *parameters* are chosen such that their expected performance is optimal within a given task context.

4.2 Action Models

Action models predict the performance or outcome of an action. Before an action is executed with certain parameters, the corresponding action model can be called with the same parameters, for instance to predict the execution duration. Action models will be discussed in more detail in Section 5.

4.3 Abstract Actions

To achieve more complex tasks, actions are combined and concatenated. There is a long tradition in using symbolic planners to generate abstract action chains for robot control. Shakey, one of the first autonomous mobile robots, used symbolic representations to determine action sequences that would achieve its goal (Nilsson, 1984). More recent examples include the work of Coradeschi and Saffiotti (2001), Beetz (2001), Cambon et al. (2004) and Bouguerra and Karlsson (2005). One of the main reasons why symbolic planning and abstract actions are of interest to robotics is that they abstract away from many aspects of the continuous belief state, so planning and replanning are faster, and more complex problems can be dealt with. Planners are good at fixing the general abstract structure of the task. Also, action sequences or action hierarchies must not be specified in advance, but are generated online, depending on the situation at hand, making the system more adaptive. Furthermore, robots can reason about plans offline before execution, to recognize and repair failures in advance (Beetz, 2000), which is preferable to encountering them during task execution. Finally, symbolic planning for robotics enables designers to specify constraints on actions symbolically (Cambon et al., 2004).

Abstract actions consist of the preconditions and effects of an action (Nilsson, 1994). They constitute the controller's abstract declarative knowledge. The effects represent the intended goal of the action. It specifies a region in the state space in which the goal is satisfied. The precondition 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 effects. Figure 4 shows two abstract actions, along with their preconditions and effects.

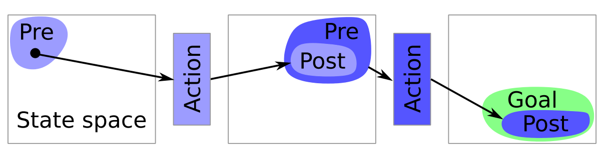


Figure 4: An abstract action chain consisting of two abstract actions.

In the system implementation, the Planning Domain Description Language (PDDL2.1) is used to describe abstract actions, goals, states and plans (Fox & Long, 2003). The declarative knowledge about the preconditions and effects (add- and delete-list) of the abstract actions in the action library are specified manually by the designer. The PDDL planner we use is the Versatile Heuristic Partial Order Planner (Younes & Simmons, 2003)[2]. VHPOP is, as all PDDL planners, a general purpose planner not specifically tailored to robot planning.

2. VHPOP can be downloaded free of cost at `http://www.tempastic.org/vhpop/`

Other work focusses on problems that need to be solved to exploit symbolic plannning in robotics, such as uncertainty, failure recovery and action monitoring (Bouguerra & Karlsson, 2005), geometric constraints (Cambon et al., 2004), and anchoring (Coradeschi & Saffiotti, 2001). The system presented in this section abstracts away from these problems to focus on the main contribution: the optimization of already generated plans. The output of VHPOP is a list of symbolic actions and causal links, of which we shall see an example in Figure 10.

4.4 System Overview

Subgoal refinement and subgoal assertion both operate on sequences of actions. In this article, these action sequences are derived from abstract action chains, which are generated by a symbolic planner. A general computational model of robot control based on symbolic plans is depicted in Figure 5, and is similar to the models proposed by Bouguerra and Karlsson (2005) and Cambon et al. (2004).

The goal is passed as an input to the system. Usually, the abstract state is derived from the belief state through a process called anchoring (Coradeschi & Saffiotti, 2001). As in the International Planning Competition, we consider a limited number of scenarios, enabling us to specify the scenarios in PDDL in advance. The name of the scenario is contained in the belief state, and the corresponding abstract state is read from a PDDL file.

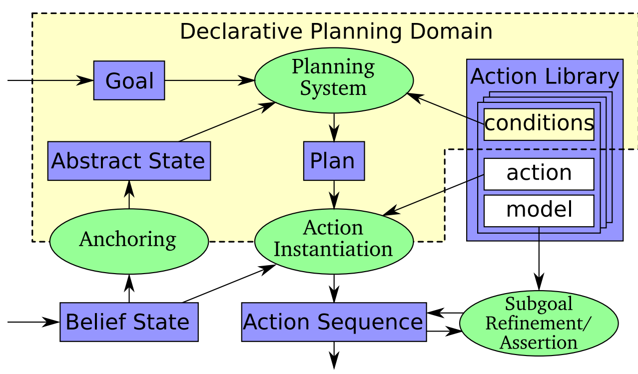


Figure 5: Computational model of robot control based on symbolic plans.

input : pddl_goal, *(represented in PDDL)*
beliefState, *(belief state with state variables)*
action_lib *(library with PDDL representations, actions, and action models)*
output : exe_action_seq *(an optimized sequences of executable actions)*

```
pddl_state = readFromFile (beliefState.scenario_name) // 'Anchoring'
pddl_plan = makePlan (pddl_state, pddl_goal, action_lib.pddl) // VHPOP
exe_action_seq = instantiateAction (pddl_plan, belief_state, action_lib.actions) // Section 6.1
exe_action_seq = assertSubgoals (exe_action_seq,belief_state, action_lib) // Section 7
exe_action_seq = refineSubgoals (exe_action_seq, action_lib.models) // Section 6.2
return exe_action_seq;
```

Algorithm 1: System Overview.

The pseudo-code for the complete system described in this article is listed in Algorithm 1. Data structures from the abstract declarative planning domain (see Figure 5) have the prefix 'pddl_'. In this section, we have presented the implementations of the functions in lines 1 and 2. In the next section, we will describe how the action models used by subgoal refinement and assertion (line 4 and 5) are learned from observed experience. The implementations of the functions in lines 3 to 5 are presented in Section 6 and 7. Note that subgoal refinement and assertion only modify existing action sequences. They do not interfere with the planning or instantiation process. This means that they are compatible with other planning systems.

Before action models can be applied online during task execution, they must be learned. This learning phase is described in the next section.

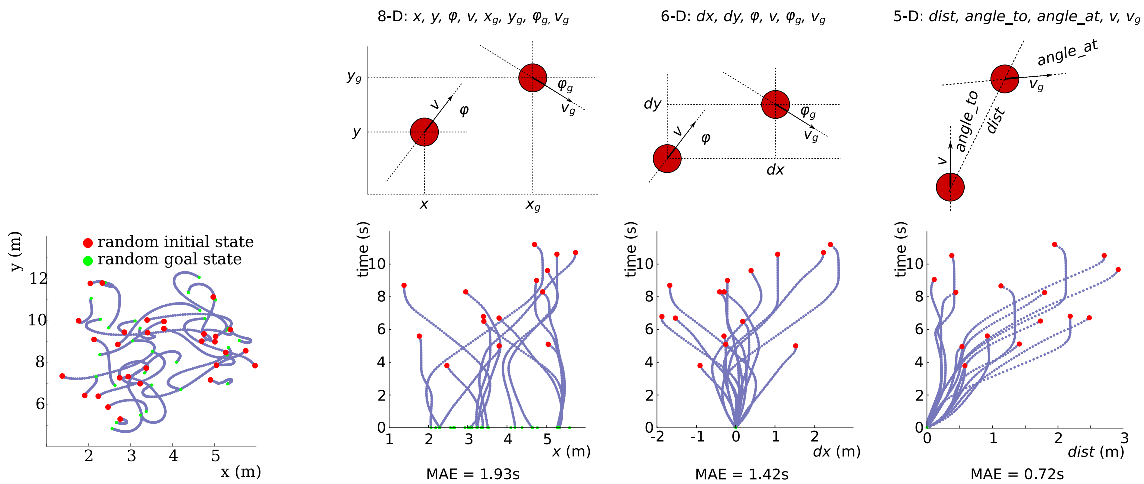
5. Learning Action Models

In this section, we describe how robots learn action models from observed experience. The advantage of learning action models over analytical methods is that 1) it is based on real experience, and therefore takes all factors relevant to performance into account. 2) many (hand-coded) actions are difficult to formalize analytically (Beetz & Belker, 2000). For instance, the exact interactions of the ball with the guide rail are difficult to predict with model-predictive control, as it is the result of the subtle interplay of many complex factors, such as the robot's dynamics, sensing capabilities, the smoothness of the ball and guide rail, parameterizations of the control program, etc. All these factors must be modeled in order to make accurate predictions. 3) analysis is sometimes impossible because the implementation the action are unknown, for instance when they have been learned. 4) although not implemented in our work, learned action models can also adapt to changing environments if trained online (Dearden & Demiris, 2005; Kirsch & Beetz, 2007). 5) it enables robots to learn models for other robots, by observing their behavior, as demonstrated by Stulp, Isik, and Beetz (2006).

5.1 Data Acquisition

Training examples are gathered by executing an action, and observing the results. In this article, robots record and learn to predict the execution duration and success of actions, given the parameterization of the action. To generate training data for an action, parameter values for the initial and goal state are sampled uniformly from the possible values the action parameters can take. The possible parameter value ranges depend on the preconditions and effects of an action, and are specified semi-automatically. The user defines ranges for the action parameters that ensure that the preconditions and effects of the action are met, and the action parameters used are uniformly sampled from these ranges. The execution of an action from an initial to a goal state is called an episode.

The running example in this section will be learning to predict the execution duration of the `goToPose` action for the simulated B21 and the Agilo robot. Figure 6(a) displays a concrete example of gathered training data with the simulated B21 robot. Here, 30 of 2948 executions of `goToPose` with random initial and goal states are shown. The total number of executions is denoted with n_e. For instance, n_e=2948 for the example above. We split this data into a training and test set. The number of examples in the training set is $N{=}\frac{3}{4}n_e$, which in the current example yields 2200 episodes.



(a) Visualization of 30 of 2948 `goToPose` executions.

(b) The original state space, and two derived feature spaces. The top figures depict the features used. The lower graphs plot the time remaining until reaching the goal against one of these features.

Figure 6: Gathering and transforming experience for `goToPose` in the kitchen domain.

Will a learning algorithm trained with these 2200 examples likely make erroneous predictions on previously unseen cases? In general, a hypothesis that is consistent with a sufficiently large training set is deemed *probably approximately correct* (PAC) (Russell & Norvig, 2003). To obtain a model that has an error of at most ϵ with probability $1-\delta$ (i.e. is PAC), the learning algorithm must be trained with at least $N \geq \frac{1}{\epsilon}(ln\frac{1}{\delta} + ln|\mathbf{H}|)$ training examples, in which $|\mathbf{H}|$ is the number of possible hypotheses. We do not use this formula to exactly compute the number of training examples needed, but rather to determine strategies to learn more accurate models with a limited amount of costly training episodes. We use three approaches:

Reduce the number of hypotheses $|\mathbf{H}|$. By exploiting invariances, we can map the data from the original direct state space to a lower-dimensional derived feature space. This limits the number of possible hypotheses $|\mathbf{H}|$. For navigation actions for instance, the translational and rotational invariances in the original 8-D state space (listed in Table 6) are exploited to transform it to a 5-D features space, depicted in Figure 6(b). The features in this 5-D state space correlate better with the performance measure. Haigh (1998) calls such features *projective*.

By exploiting the invariances, we are reducing the dimensionality of the feature space, which leads to a decrease of $|\mathbf{H}|$. This equation specifies that with lower $|\mathbf{H}|$, fewer training examples are needed to learn a PAC model. By the same reasoning, more accurate models (i.e. lower ϵ) can be learned on lower dimensional feature spaces, given the same amount of data.

We have experimentally verified this by training the model tree learning algorithm (to be presented in Section 5.2) with data mapped to each of the three different feature spaces in Figure 6(b). For each feature space, the model is trained with N=2200 of the n_e=2948 executed episodes. The Mean Absolute Error (MAE) of each of these models is determined on the separate test containing the remaining episodes[3]. As can be seen in Figure 6(b), the MAE is lower when lower dimensional feature spaces are used. Of course, this lower dimensionality should not be achieved by simply discarding informative features, but rather by composing features into projective features by exploiting invariances.

Increase the number of training examples N by including intermediate examples. Instead of using only the first initial example of each episode (large red dots in Figure 6(b)), we could also include all the intermediate examples until the goal state. As these are recorded at 10Hz, this yields many more examples, and higher N. However, one characteristic of projective features is that they pass through the point of origin, as can be seen in the right-most graph in Figure 6(b). Therefore, including all intermediate examples will lead to a distribution that is skewed towards the origin. This violates the stationarity assumption, which poses that the training and test set must be sampled from the same probability distribution. Most learning algorithms trained with this abundance of data around the origin will be biased towards states that are close to the goal, and will tend to predict these states very accurately, at the cost of inaccurate prediction of states further from the goal.

Therefore, we include only the first n_i examples of each episode. This means that the number of training examples is roughly $n_e \cdot n_i$ instead of just n_e, but still represents the original distribution of initial states. Since the best value of n_i is not clear analytically, we determine it empirically, as described by Stulp (2007)

Track the error ϵ empirically. Instead of defining ϵ in advance, we compute the Mean Absolute Error (MAE) over time as more data is gathered, and determine when it stabilizes visually. At this point, we assume that N is sufficiently large, and stop gathering data.

5.2 Tree-based Induction

Learning algorithms that are used for learning robot action models from observed experience include neural networks (Buck, Beetz, & Schmitt, 2002b), and tree-based induction (Belker, 2004; Haigh, 1998). We have shown that there is no significant difference in the accuracy of action models learned with neural networks or tree-based induction (Stulp et al., 2006). However, decision and model trees have the advantage that they can be converted into sets of rules, which can then be visually inspected. Furthermore, model trees can be optimized analytically as described in Section 6.2. For these reasons, we will focus only on decision and model trees in this article.

Decision trees are functions that map continuous or nominal input features to a nominal output value. The function is learned from examples by a piecewise partitioning of the feature space. One class is chosen to represent 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. A line is fitted to the data in each partition with linear regression.

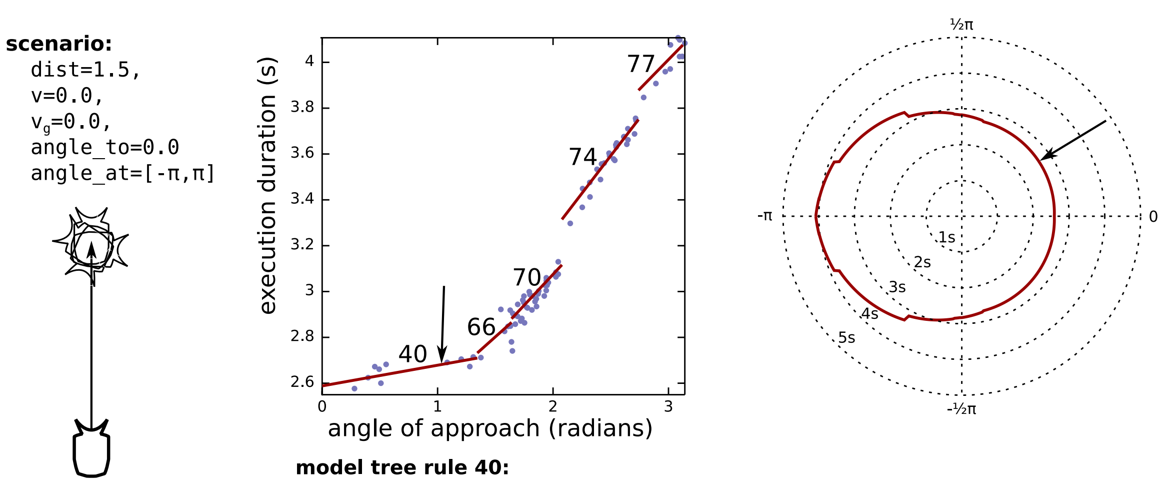
3. We prefer the MAE over the Root Mean Square Error (RMSE), as it is more intuitive to understand, and the cost of a prediction error is roughly proportional to the size of the error. There is no need to weight larger errors more.

This linear function interpolates between data in the partition. This enables model trees to approximate continuous functions. For more information on decision and model trees, and how they are learned with tree-based induction, we refer to (Quinlan, 1992; Stulp et al., 2006; Wimmer, Stulp, Pietzsch, & Radig, 2008). In our implementation, we use WEKA (Witten & Frank, 2005) to learn decision and model trees, and convert its output to a C++ function.

5.3 Action Model: Execution Duration Prediction

To visualize action models learned with model trees, an example of execution duration prediction for a specific situation is depicted in Figure 7. This model for the `goToPose` action in the soccer domain (real robots) is learned from n_e=386 episodes. The first n_i=20 examples per episode are used. The features used are *dist*, *angle_to*, *angle_at*, v and v_g, as depicted in Figure 6(b).

In the situation depicted in Figure 6(b), the variables *dist*, *angle_to*, v, and v_g are set to 1.5m, 0°, 0m/s, and 0m/s respectively. The model is much more general, and predicts accurate values for any *dist*, *angle_to*, v, and v_g; these variables are fixed for visualization purposes only. For these fixed values, Figure 7 shows how the predicted time depends on *angle_at*, once in a Cartesian, once in a polar coordinate system.



model tree rule 40:
```
if (1.7 > dist > 1.3) ∧ (angle_to < 1.19) ∧ (angle_at < 1.36)
  then time = 1.12*dist + 0.090*angle_at + 0.011*angle_to + 0.91;
```

Figure 7: An example situation, two graphs of time prediction for this situation with varying *angle_at*, and the model tree rule for one of the line segments.

In the Cartesian plot there are five line segments. This means that the model tree has partitioned the feature space for *dist*=1.5m *angle_to*=0°, v=0m/s, and v_g=0m/s 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. Arrows in the graphs indicate the linear model that corresponds to this rule.

5.3.1 Empirical Evaluation

The different domains and actions for which action models of execution duration are learned are listed in the first two columns of Table 1. The subsequent columns list the number of episodes executed to gather data for the training set $\frac{3}{4}n_e$, the mean execution duration per episode $\overline{t}$, the total duration of data gathering for the training set $\overline{t} \cdot \frac{3}{4}n_e$, as well as the model's error (MAE) on a separate test set with the remaining $\frac{1}{4}n_e$ episodes. Note that the final model stored in the action library is trained with data from all n_e episodes. In the next sections, we demonstrate that these action models are accurate enough to enable a significant improvement of action execution.

Domain	Action	$\frac{3}{4}n_e$	$\overline{t}$ (s)	$\overline{t} \cdot \frac{3}{4}n_e$ (h:mm)	MAE (s)
Soccer (real)	`goToPose`	290	6.4	0:31	0.32
	`dribbleBall`	202	7.7	0:26	0.43
Soccer (simulated)	`goToPose`	750	6.2	1:18	0.22
	`dribbleBall`	750	7.4	1:32	0.29
Kitchen	`goToPose`	2200	9.0	5:45	0.52
	`reach`	2200	2.6	1:38	0.10
Arm control	`reach`	1100	2.9	0:53	0.21

Table 1: List of actions and their action model statistics.

5.4 Action Model: Failure Prediction

The simulated soccer robots also learn to predict failures in approaching the ball with the `goToPose` action. A failure occurs if the robot collides with the ball before the desired state is achieved. The robots learn to predict these failures from observed experience. To acquire experience, the robot executes `goToPose` a thousand times, with random initial and goal states. The ball is always positioned at the destination state. The initial and goal state are stored, along with a flag that is set to `Fail` if the robot collided with the ball before reaching its desired position and orientation, and to `Success` otherwise. The feature space is the same as for learning the temporal prediction model of `goToPose`, as listed in Table 7.

The learned tree, and a graphical representation, are depicted in Figure 8. The goal state 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, the robot always clumsily stumbles into the ball, long before reaching the desired orientation. Approaching the ball is fine from any state in the green area labeled `S`.

5.4.1 Empirical Evaluation

To evaluate the accuracy of this model, the robot executes another thousand runs. The resulting confusion matrix is depicted in Table 2. The decision tree predicts collisions correctly in almost 90% of the cases. The model is quite pessimistic, as it predicts failure

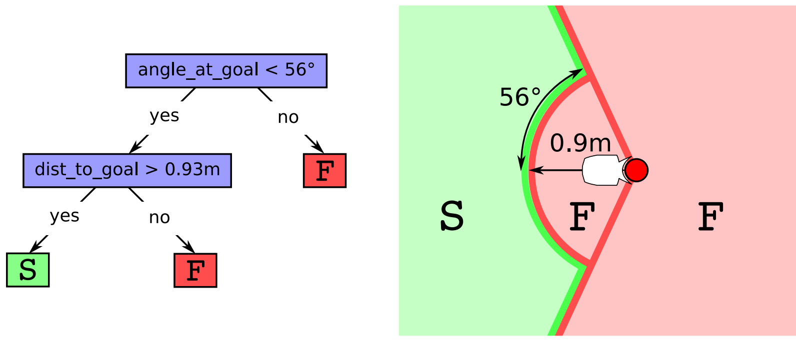


Figure 8: The learned decision tree that predicts whether a collision will occur.

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. This pessimism is actually no coincidence; it is caused because a cost matrix that penalizes incorrect classification of `Fail` more than it does `Success` is passed to the decision tree (Witten & Frank, 2005).

		Observed			Total
		`Fail`	`Success`		Predicted
Predicted	`Fail`	51%	10%	→	61%
	`Success`	1%	38%	→	39%
		↓	↓	↘	
	Total Observed	52%	48%		89%

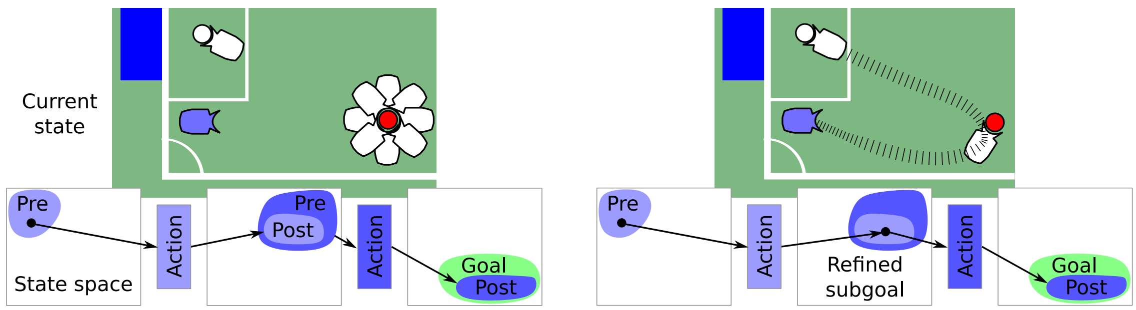
Table 2: Confusion matrix for ball collision prediction.

6. Subgoal Refinement

When it comes to elegant motion, robots do not have a good reputation. Jagged movements are actually so typical of robots that people trying to imitate robots will do so by executing movements with abrupt transitions between them. Figure 1(b) demonstrates an abrupt transition that arises when approaching the ball to dribble it to a certain location. Such jagged motion is not just inefficient and aesthetically displeasing, but also reveals a fundamental problem that inevitably arises from the way robot controllers and actions are designed and reasoned about.

Figure 9(a) depicts the abstract action chain of the scenario, with preconditions and effects represented as subsets of the entire state space. Note that the intersection of preconditions and effects at the intermediate subgoal contain many intermediate states, eight of which are also depicted in the scenario in Figure 9(a). In this set, all variables are equal, except the angle with which the ball is approached. This action parameter is therefore called *free*.

As discussed in Section 1, the reason for abrupt transitions is that although the preconditions and effects of abstract actions justly abstract away from action parameters that do not influence the high-level selection of actions, these same free parameters might influence the performance of actually executing them. For instance, the angle of approach is abstracted away when selecting the actions, although it obviously influences the execution duration. The first step in subgoal refinement is therefore to determine free action parameters in a sequence of abstract actions.



(a) Abstract action chain before subgoal refinement. (b) Abstract action chain optimized with subgoal refinement.

Figure 9: Computational model of subgoal refinement.

In contrast to robot motion, one of the impressive capabilities of animals and humans is their ability to perform sequences of actions efficiently, and with seamless transitions between subsequent actions. It is assumed that these typical patterns are those that minimize a certain cost function (Wolpert & Ghahramani, 2000; Schaal & Schweighofer, 2005). So, in nature, fluency of motion is not a goal in itself, but rather an emergent property of time, energy and accuracy optimization.

Subgoal refinement exploits free action parameters in a similar way. Since all the states in the intermediate set lead to successful execution of the action sequence, we are free to choose whichever state we want. Execution will succeed for any value for the free angle of approach. As we saw in Figure 1 some values are better than others, with respect to the expected execution duration. Therefore, subgoal refinement determines values for the free action parameters that minimize the predicted execution duration of the entire action sequence. The prediction is made by action models.

The behavior shown after applying subgoal refinement in Figure 1(c) has better performance, achieving the ultimate goal in less time. A pleasing side-effect is that it exhibits seamless transitions between actions.

We will now proceed by explaining how executable actions are instantiated from the abstract PDDL plans generated by VHPOP, and how free action parameters arise in this process. Then, we describe how subgoal refinement optimizes these free action parameters.

6.1 Action Instantiation and Free Action Parameters

Causal links specify which action was executed previously to achieve an effect which meets the precondition of the current action. A chain of such abstract actions represents a valid plan to achieve the goal, an example of which is shown in Figure 10. The next step in our system is to map such plans to the executable actions in the action library. This process is also known as *operator instantiation* (Schmill et al., 2000).

```
Initial 0 : (robot pos1) (ball pos2) (final pos3)

Step 2   : (approachball pos1 pos2)
          0   -> (ball pos2)
          0   -> (robot pos1)

Step 1   : (dribbleball pos2 pos3)
          2   -> (atball pos2)

Goal     :
          0   -> (final pos3)
          1   -> (atball pos3)
```

Figure 10: The output of VHPOP is a PDDL plan with causal links.

PDDL plans are instantiated with executable actions by first extracting symbolic actions and causal links in the plan, and then instantiating the symbolic actions one by one, as listed in Algorithm 2. For each symbolic action, the executable action is retrieved by its name (line 5), after which its parameters are requested (line 6). The next step is to determine the parameter values of the executable action, by considering the corresponding symbolic parameters of the PDDL plan. The correspondence between the executable action parameter and a symbolic action parameter is determined based on an index in the executable action parameter (line 8).

The symbolic parameters themselves have no meaning in the belief state. They are just labels used in the PDDL plan. However, causal links define predicates over these labels which *do* have a meaning in the belief state. These predicates are therefore retrieved (line 9), and used to extract the correct values from the belief state (line 10).

Mapping symbolic predicates to continuous values is done in the belief state, with the call made in line 10. If the predicate holds in the current belief state, which is the case if it starts with a '`0`' (the initial state is considered the first 'action'), it simply retrieves the value. For '`0robot`' and '`x`' it would return the x-coordinate of the current position of the robot. Predicates that do not hold in the current state can also constrain values. For instance, the `atball` predicate restricts the translational velocity between 0 and 0.3m/s. If predicates impose no such constraints, default values for the parameter types are returned. For instance, the values of x-coordinates must be within the field, and angles are always between -π and π. If several predicates hold, the ranges and values they return are composed.

6.2 Optimizing Free Action Parameters

Mapping abstract PDDL plans to executable actions often leads to free action parameters. Humans also have a high level of redundancy in their actions. As Wolpert and Ghahramani (2000) note, "there are many ways you can bring a glass of water to your lips, and some are sensible and some are silly". The reason why we typically witness stereotypical 'sensi-

```
    input   : pddl_output (the output of VHPOP, see Figure 10 for an example)
    output  : exe_actions (a parameterized sequence of executable actions)
 1  pddl_actions = parseActions(pddl_output);
 2  pddl_links = parseCausalLinks(pddl_output);
    // For the example in Figure 10, the following now holds:
    // pddl_actions = [(approachball pos1 pos2),(dribbleball pos2 pos3)]
    // pddl_links = {pos3=[0center,1atball], pos1=[0robot], pos2=[0ball,2atball]}
 3  exe_actions = {};
 4  foreach pddl_action in pddl_actions do
 5      exe_action = getAction(pddl_action.name) // e.g. exe_action = approachBall
 6      exe_params = exe_action.getParameters() // then exe_params = [x0,y0,...]
 7      foreach exe_par in exe_params do
 8          pddl_par = pddl_action.params[exe_par.index] ;
            // e.g. if exe_par = x0, then exe_par.index = 0 and pddl_par = pos1
 9          pddl_predicates = pddl_links[pddl_par] ;
            // e.g. if pddl_par = pos1, then pddl_predicates = [0robot]
10          value = beliefState.getValue(exe_par.name, pddl_predicates) ;
11          exe_action.setParameter(exe_par, value);
12      end
13      exe_actions.add(exe_action);
14  end
    // For the example in Figure 10 and Figure 9(a), the following now holds:
    // exe_actions = [ approachBall(x=0,y=1,φ=0,v=0, xg=3,yg=1,φg=[-π,π],vg=[0,0.3]),
    // dribbleBall(x=3,y=1,φ=[-π,π],v=[0,0.3], xg=1,yg=3,φg=2.6,vg=0) ]
15  return exe_actions;
```

Algorithm 2: Action Instantiation. Implementation of line 3 of Algorithm 1.

ble' and fluent movement is because the redundancy is exploited to optimize 'subordinate criteria' (Schaal & Schweighofer, 2005), or 'cost functions' (Wolpert & Ghahramani, 2000), such as energy efficiency or variance minimization.

We also take this approach, and optimize free parameters in action sequences with respect to the expected execution duration. To optimize an action sequence, the system will have to find values for the free action parameters such that the overall predicted execution duration of the sequence is minimal. This overall performance is estimated by simply summing over the action models of all actions that constitute the sequence (Stulp & Beetz, 2005).

Let us again take the example action sequence below line 14 in Algorithm 2 as an example. In Figure 11, Figures 1 and 7 are combined. The first two polar plots represent the predicted execution duration of the two individual actions for different values of the free angle of approach. The overall duration is computed by simply adding those two, as is depicted in the third polar plot. The fastest time to execute the first `approachBall` can be read in the first polar plot. It is 2.5s, for an angle of approach of 0.0 degrees, as indicated in the first plot. However, the total time for executing both `approachBall` and `dribbleBall` for this angle is 7.4s, because the second action takes 4.9s. The third plot clearly shows that this is not the optimal overall performance. The minimum is actually 6.5s, for an angle of 50°. Beneath the polar plots, the situation of Figure 1 is repeated, this time with the predicted performance for each action.

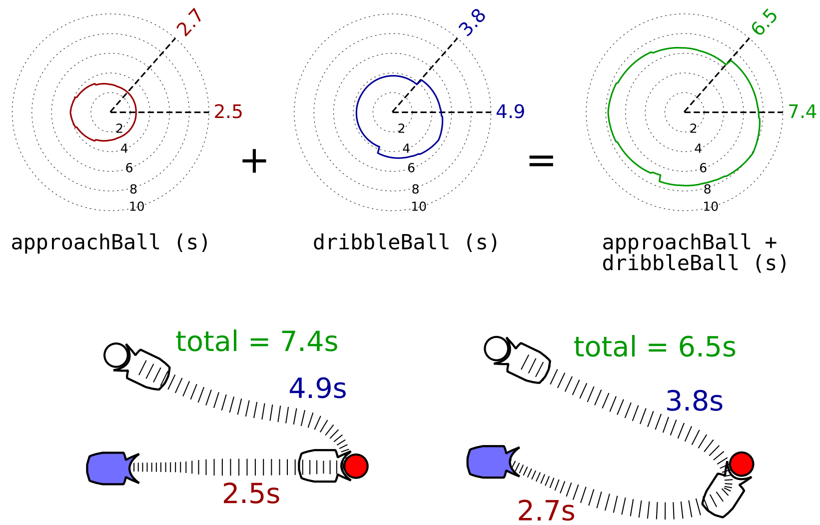


Figure 11: Selecting the optimal subgoal by finding the optimum of the summation of all action models in the chain.

For reasons of clarity, only one parameter is optimized in this example, and we simply 'read' the minima from the plot. Online however, the robots must be able to determine this minimum automatically, possibly with several free action parameters and resulting high-dimensional search spaces. The next sections describe two optimization methods.

6.2.1 Analytical Optimization of Model Trees

In Figure 7, the action model clearly consists of line segments in the 1-dimensional feature space. In general, model trees partition the d-dimensional feature space into k partitions, and represent the data in each partition with a d-dimensional hyperplane.

This representation allows an analytical minimization of model trees. The solution idea is that the minimum of a hyperplane can be found quickly by determining the values at its corners, and taking the corner with the minimum value. This procedure should be repeated for all k hyperplanes, which leads to k corner minima. The global minimum can then be determined by choosing the minimum of all 'minimal corners'. This novel analytical method has a complexity of $O(kd)$, in which k is the number of hyperplanes (i.e. the number of rules, or leaves in the model tree), and d the number of dimensions. Therefore, this method does not suffer from the curse of dimensionality, as the complexity linearly depends on the number of dimensions, instead of exponentially.

Determining the minimum of two or more model trees is done by first merging the model trees into one, and then determining the minimum of the new model tree. Unfortunately, there are cases in which model trees cannot be merged, for instance when there is a non-linear mapping between action parameters and the features derived from them. In these cases, analytical optimization of summations of trees is not possible, and a genetic algorithm is used. The implementation of the analytical optimization, model tree merging, as well as a formalization of the special cases in which model trees cannot be merged are presented by Stulp (2007).

6.2.2 Optimization with a Genetic Algorithm

In the cases when model trees cannot be optimized analytically, we use a genetic algorithm for optimization. Note that the problem of optimizing the action sequence optimization has been simplified to a straightforward function optimization in which the search space is determined by the free action parameters and their ranges, and the target function whose minimum is sought is determined by the model trees, as in Figure 11. Since model trees have many discontinuities and can therefore not be differentiated, we have chosen a genetic algorithm for optimization, as it can be applied well to such non-differentiable non-continuous functions. Figure 12 depicts how optimization with a genetic algorithm (Goldberg, 1989) is integrated in the overall system. At the top, an instantiated action sequence with bound and free action parameters is requested to be optimized.

Note that the parameters are labeled with an identification number (ID). Action parameters have the same ID if the symbolic parameters from which they were derived are the same, see line 8 in Algorithm 2. For reasons of brevity, ID allocation is not included in Algorithm 2. The IDs reflect that certain parameters in different actions always have the same value, as they are identical. For instance, the goal orientation (ϕ_g) of the `approachBall` is equivalent to the initial orientation (ϕ) of `dribbleBall`. Therefore they share the ID '13'. Note that this is a free action parameter.

The first step is to partition the action parameters in the action sequence into two sets: one set contains action parameters that are bound to a certain value during instantiation, and the other set contains free action parameters, along with the range of values they can take. Note that action parameters with the same ID are only stored once in these sets, as they should have the same value.

Each free action parameter is represented as a floating point gene on a chromosome. The number of chromosomes in the population is the number of free parameters multiplied by 25. The chromosomes in the initial population are initialized with random values from their respective ranges. The standard genetic algorithm (GA) loop is then started. The loop halts if the best fitness has not changed over the last 50 generations, or if 500 generations are evaluated. The optimal values of the free parameters are then also bound in the action sequence.

For a chromosome, the predicted execution duration is determined by calling the action models with the fixed values from the set of bound parameters, and the values of the free parameters represented in the chromosome. For fitness proportionate selection, the fitness should be a non-negative number which is larger with increased fitness. Therefore, for each chromosome c the fitness f is computed with $f_c = t_{max} + t_{min} - t_c$, where t_{max} and t_{min} are the maximum and minimum execution duration over all chromosomes respectively.

Our implementation of the genetic algorithm uses elitarianism (2% best individuals passes to the next generation unmodified), mutation (on the remaining 98%), two-point crossover (on 65% of individuals), and fitness proportionate selection (the chance of being selected for crossover is proportionate to an individual's fitness). To test and evaluate our GA implementation in C++, we first applied it to several optimization benchmarks, such

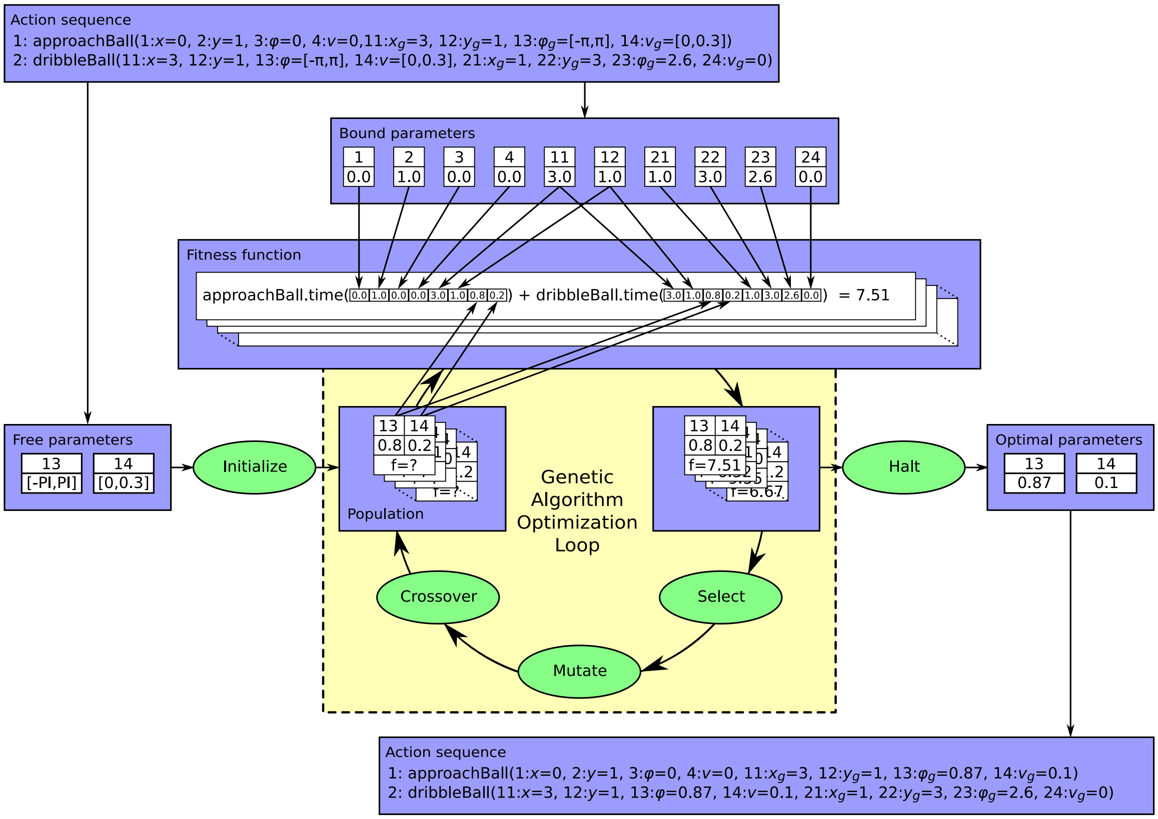


Figure 12: Optimization in subgoal refinement with a genetic algorithm. Implementation of line 5 of Algorithm 1.

as the De Jong's function, Schwefel's function and Ackley's Path function. The results and optimization times are reported by Koska (2006).

In the subgoal refinement scenarios to be presented in Section 6.3, the optimization time is usually small in comparison to the gain in performance. In the most complex kitchen scenario, where up to 4 actions with more than 10 action parameters are optimized, our implementation of the GA still takes less than 0.5s to get a good result.

6.3 Empirical Evaluation

In this section, we introduce the different scenarios in which the results of of applying subgoal refinement are evaluated. In the robotic soccer domain, the action sequence to be optimized is the `approachBall` action, followed by a `dribbleBall` action, as in Figure 1. The free action parameters at the intermediate state are the angle of approach and the translational velocity.

To evaluate the effect of subgoal refinement in the service robotics domain, two scenarios are tested. In the first scenario, the goal is to put a cup from one table to another, which is achieved by a sequence of navigation and grasping actions. In each evaluation episode, the topology of the environment in each scenario stays the same, but the initial robot position, the tables, and the cups are randomly displaced. Scenario 2 is a variation of Scenario 1, in which two cups had to be delivered.

The kitchen scenarios have many free action parameters. Because preconditions usually bind either navigation (`goToPose`) *or* manipulation (`grip` or `put`) actions but never both (they are independent), one of these action parameter sets is always free. Furthermore, the distance of the robot to the table in order to grab a cup must be between 40 and 80cm (as fixed in the precondition of `grip`). This range is another free parameter. As in the soccer domain, the velocity and orientation at waypoints are also not fixed, so free for optimization as well.

In the arm control domain, sequences of reaching movements are performed. Because this particular task does not require abstract planning, we did not use VHPOP. For demonstration purposes, we had the arm draw the first letter of the first name of each author of the paper by Stulp, Koska, Maldonado, and Beetz (2007), and chose 4/5 waypoints accordingly. Figure 13(a) shows the PowerCube arm, which is attached to a B21 robot, drawing an 'F'. To draw these letters, only two of the six degrees of freedom of the arm are used. The free action parameters are the angular velocities at these waypoints.

Table 3 lists the results of applying subgoal refinement to the different domains and scenarios, where a is the number of actions in the sequence, and n is the number of episodes tested. The baseline with which subgoal refinement is compared is a greedy approach, in which the next subgoal is optimized with respect to the execution duration of *only* the current action. In this case, we say the horizon h of optimization is 1. The downside of the greedy baseline is that it also depends on the accuracy of the action model. However, we chose this as a baseline, because setting all free action parameters to zero certainly leads to worse execution times, and optimizing them manually introduces a human bias.

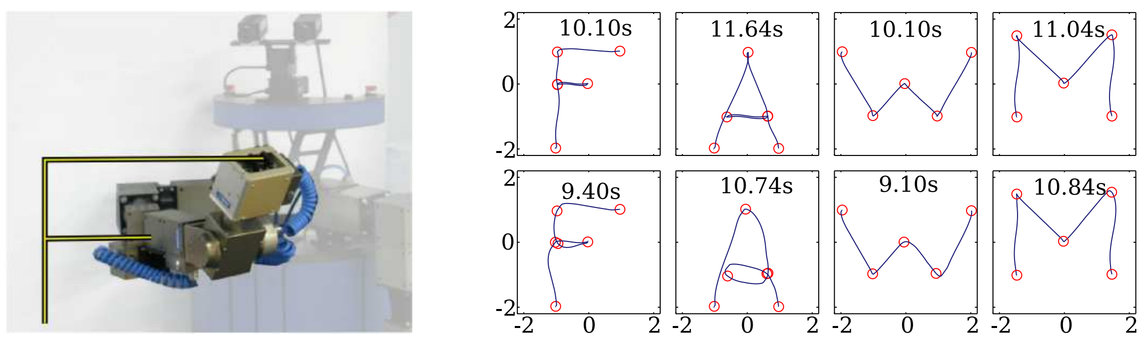
Scenario	a	n	$\overline{t_1}$	$\overline{t_2}$	$\overline{t_{2/1}}$	p
Soccer (real)	2	100	10.6s	9.9s	6.1%	0.00
Soccer (simulated)	2	1000	9.8s	9.1s	6.6%	0.00
Kitchen (scenario 1)	4	100	46.5s	41.5s	10.0%	0.00
Kitchen (scenario 2)	13	100	91.7s	85.4s	6.6%	0.00
Arm control	4-5	4	10.6s	10.0s	5.7%	0.08

Table 3: Subgoal refinement results.

The execution time of a single action is denoted t_h^e, which has two indices referring to the horizon and the episode. For instance t_1^{64} refers to the total execution time of the 64th episode, performed with a horizon of 1. The mean overall execution duration over all episodes is denoted $\overline{t_1}$, and computed with $\overline{t_1} = \frac{1}{n}\sum_{e=1}^{n} t_1^e$. Since subgoal refinement optimizes the execution duration of the current *and* next action, it has a horizon of 2. The fourth column lists the mean overall execution duration with subgoal refinement $\overline{t_2}$, and computed with $\overline{t_2} = \frac{1}{n}\sum_{e=1}^{n} t_2^e$.

The equation $\overline{t_{2/1}} = \frac{1}{n}\sum_{e=1}^{n}(1 - t_2^e/t_1^e)$ is used to compute the mean improvement over all episodes achieved with subgoal refinement. The p-value of the improvement is computed using a dependent t-test with repeated measures, as each episode is performed twice, once with, and once without subgoal refinement. A significant improvement occurs in all but one domain.

To visualize the qualitative effect of applying subgoal refinement, the results from the arm control domain are depicted in Figure 13(b). The angular velocities are set to zero (upper row) or optimized with subgoal refinement (lower row). The axes represent the angles of the two joints. This figure demonstrates that the trajectories are smoother with subgoal refinement: the arm mostly draws one long stroke, rather than discernible line segments. Since the arm control domain is mainly included for visualization purposes, there are only four test runs. For this reason the overall improvement is not significant ($p > 0.05$).



(a) The B21 robot drawing an 'F' with its PowerCube arm.

(b) Drawing letters without (upper row) and with (lower row) subgoal refinement. With refinement, letters are drawn faster and smoother.

Figure 13: Arm control experiment.

Although optimizing execution duration leads to smoother motion in this domain, smooth human arm motion arises from variability rather than time optimization (Harris & Wolpert, 1998; Simmons & Demiris, 2004). In this article, the main goal is not to explain or model human motion, but rather to demonstrate the effects of optimizing sequences of actions.

6.3.1 Sequences with More Actions

So far, we have seen optimization with horizons of $h = 1$ (greedy) and $h = 2$. The standard approach with $h = 2$ can easily be extended, so that subgoal refinement optimizes the execution duration of the next $h > 2$ actions. The higher the horizon h, the more subgoal refinement is preparing for actions further in the future.

To evaluate the effect of optimizing more than two actions, sequences of four and more actions are optimized using subgoal refinement with different horizons. Two scenarios are used: a simulated soccer scenario in which the robot has to traverse four waypoints with the `goToPose` action, and the two kitchen scenarios. After each action execution, the subgoals between subsequent actions are recomputed with the same horizon, or less if the number of remaining actions is smaller than h. The results are summarized in Table 4. The first row represents the baseline greedy approach with $h = 1$, and the second row represents the results reported so far with $h = 2$. The next two rows list the results of optimizing 3 and 4 action execution durations. Again, the reported times represent the execution duration of the entire action sequence, averaged over 100 episodes.

horizon	Soccer			Kitchen (Scen.1)			Kitchen (Scen.2)		
	$\sum$	Imp.	p-value	$\sum$	Imp.	p-value	$\sum$	Imp.	p-value
$h = 1$	22.7			46.5			91.7		
$h = 2$	20.3	10.6%	0.000	41.5	10.0%	0.000	85.4	6.6%	0.041
$h = 3$	20.2	0.7%	0.001	40.6	1.5%	0.041	85.3	0.1%	0.498
$h = 4$	20.2	0.2%	0.053	-	-	-	-	-	-

Table 4: Effect of the subgoal refinement horizon h on performance improvement.

In all three scenarios, there is a substantial improvement from $h = 1$ to $h = 2$, but from $h = 2$ to $h = 3$ the improvement is marginal or insignificant. When executing an action in these scenarios, it is apparently beneficial to prepare for the next action, but not for the action after that. This insight is also the main motivation behind receding horizon control, as described in Section 3.2. However, there are also some important differences between RHC and subgoal refinement. First of all, optimal control plans and optimizes motor commands of fixed duration, whereas subgoal refinement does so for durative actions of varying duration, and at a higher level of temporal abstraction. Furthermore, RHC optimizes the first H_p motor commands, whereas subgoal refinement optimizes action parameters at partially fixed subgoals, and is not concerned with the actual motor commands needed to reach the subgoals. The planner can therefore fix the general structure of the plan, rather than committing to only the first few steps. Finally, optimal control assumes that precise analytic models of the actions and the systems behavior are available. For many robotic systems, as well as humans, this does not hold. However, the lack of analytic models does not keep us from acquiring models from experience. By learning action models, our system is also flexible enough to acquire action models for changing actions, or actions for which no model can be acquired through analysis.

6.3.2 Predicting Performance Decrease

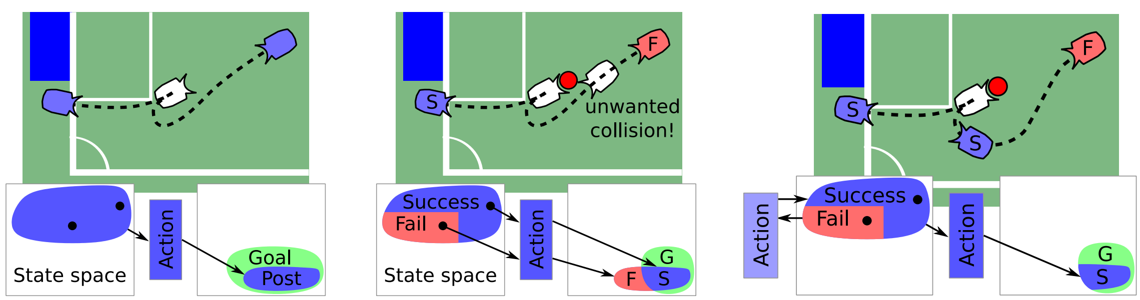
There are cases in which subgoal refinement does not have an effect. In the ball approach scenario for instance, if the robot, the ball and the final destination are perfectly aligned, there is not much to do for subgoal refinement, as the greedy approach already delivers the optimal angle of approach: straight towards the ball. On the contrary, refining subgoals in these cases might put unnecessary constraints on the execution. Due to inaccuracies in the action models and the optimization techniques, it is sometimes even the case that the greedy approach does slightly better than subgoal refinement.

To evaluate these effects, 1000 episodes were executed in simulation with both $h = 1$ and $h = 2$. Although the overall improvement with $h = 2$ was 6.6% (see Table 3), 160 of 1000 episodes actually lead to an increased execution duration. These episodes were labeled `-`, and the remaining with `+`. We then trained a decision tree to predict this nominal value. This tree yields four simple rules which predict the performance difference correctly in 87% of given cases. The learned decision tree is essentially an action model too. Rather than predicting the outcome of an individual action, it predicts the outcome of applying action models to actions.

We performed another 1000 test episodes, as described above, but only applied subgoal refinement if the decision tree predicted that applying it would yield a better performance. The overall improvement was so raised from 6.6% to 8.6%.

7. Subgoal Assertion

In practice, learning actions hardly starts from scratch, and knowledge about previously learned actions is transfered to novel task contexts. For both humans and robots for instance, approaching a ball is very similar to navigating without considering the ball. On an abstract level, both involve going from some state to another on the field, as in Figure 14(a), and both should be implemented to execute as efficiently and fast as possible. However, there are also slight differences between these two tasks. When approaching the ball it is important to not bump into it before achieving the desired state, as depicted in Figure 14(b).



(a) Original scenario. (b) Refined goal, learned with condition refinement. (c) Subgoal assertion based on condition refinement.

Figure 14: Computational model of condition refinement and subgoal assertion.

Figure 14 illustrates these small differences. In the first scenario, there is no ball, and the effects of both `goToPose` actions satisfy the goal, so both actions achieve the goal. Figure 14(b) is basically the same scenario, with the added requirement that the robot must be in possession of the ball at the goal state. The same `goToPose` action is used to achieve these goals. Since `goToPose` is not aware of the ball, it often collides with the ball before achieving the desired state. The new goal of approaching the ball is essentially a refined subset of the former goal of simply navigating there. When executing `goToPose`, the robot to the left succeeds in approaching the ball, but the robot to the right does not, as it bumps into the ball beforehand. This is the case because the effects of `goToPose` no longer satisfy the refined goal, as is depicted below the scenario.

The effects of `goToPose` are now partitioned into a subset which does satisfy the new refined goal, and a subset which does not. These are represented with blue (S) and red (F) respectively. Analogously, the preconditions are partitioned into a subset `Success` which leads to a final state which is in the subset of the effects that satisfy the refined goal, and a subset `Fail` for which this is not the case. Because the effects, and consequently, preconditions of an action are refined for a new task, we call this *condition refinement*.

Once the refined precondition of a novel goal is known, it is easy to determine if a particular initial state will lead to a successful execution or not. If it does, the action is executed as is. For instance, the robot to the left can simply execute the `goToPose` action, as it is in the refined precondition. As we shall see, the `goToPose` action can be used to approach the ball successfully almost half the time.

However, the robot to the right cannot simply use `goToPose` to approach the ball. This robot now needs a novel action, e.g. `approachBall`, that enables it to go from any of the states in the `Fail` to the refined goal. Or does it? Instead, the `goToPose` action is used again, to take the robot from the `Fail` subset to the `Success` subset. Once this is done, a `goToPose` action that *will* succeed at approaching the ball is executed.

The key to reuse is therefore being able to predict when an action will fail, and when it will succeed at a novel task. When it is predicted to succeed, the action is executed as is. If the action will fail, another action should be executed beforehand, such that the robot ends up in a state from which the action *will* succeed, as depicted in Figure 14(c). This intermediate state between actions is a new subgoal. This approach is therefore called subgoal assertion.

```
input   : exe_actions (a sequence of (partially) instantiated actions)
output  : exe_actions2 (a sequence of (partially) instantiated actions with asserted subgoals)
exe_actions2 = {};
foreach exe_action in exe_actions do
    switch exe_action.name do
        case 'goToPose'
            exe_actions2.add (exe_action) // Subgoal assertion never needed for this action
        end
        case 'approachBall'
            // Get the parameters related to the 'from' and 'to' states.
            // Uses the same indexing scheme as in lines 6-8 of Algorithm 2..
            exe_params0 = exe_action.getParameters(0);
            exe_params1 = exe_action.getParameters(1);
            if goToPose.approachBallSuccess(exe_params0, exe_params1) then
                // goToPose will do the job, subgoal assertion not needed
                exe_actions2.add (new goToPose(exe_params0, exe_params1));
            else
                // exe_params2 is set to the default ranges of the action parameters of goToPose.
                // Same as in lines 10-11 of Algorithm 2.
                exe_params2 = ...;
                exe_actions2.add (new goToPose(exe_params0, exe_params2));
                exe_actions2.add (new goToPose(exe_params2, exe_params1));
            end
        end
        ...
    end
end
return exe_actions2;
```

Algorithm 3: Implementation of line 4 of Algorithm 1.

7.1 Integration in the Overall System

Subgoal assertion takes a sequence of actions, and returns the same sequence *with* asserted subgoal that are needed to assure successful execution, as listed in Algorithm 3. The main loop goes through all actions, and leaves `goToPose` actions untouched. `approachBall` has no implementation itself, and is replaced by `goToPose` actions. Only one `goToPose` is needed if it is predicted to succeed at approaching the ball. This is the case if the initial state is in the `Success` subset in Figure 14(b). Determining these subsets manually is a difficult task, due to complex interactions between the dynamics and shape of the robot, as well as the specific characteristics of the action. Therefore, these subsets are learned with a decision tree, as described in Section 5.4.

If a success is predicted, one `goToPose` is executed as is, with the same parameters as the `approachBall` action. If it is predicted to fail, a subgoal is asserted (exe_params2), and inserted between two `goToPose` actions. The action parameters exe_params2 initially receive default ranges. All parameters in exe_params2 are free, and are optimized with subgoal refinement. This immediately follows subgoal assertion, as listed in Algorithm 1. This ensures that the values for exe_params2 minimize the predicted execution duration, and that the transition between the two `goToPose` actions is smooth.

One issue remains open. The intermediate goal between the actions must lie within the `Success` subset in Figure 14, which for the ball approach task is any position in the green area to the left in Figure 8. This requirement puts constraints on the values of exe_params2. It must be ensured that the optimization process in subgoal refinement only considers states from the `Success` subset of the refined precondition of the second `goToPose` action. Therefore, the action model for this action is modified so that it returns an INVALID flag for these states. This approach has been chosen as it requires little modification of the optimization module. Chromosomes that lead to an invalid value simply receive a low fitness.

goToPose.`executionDurationApproachBall` $(x,y,\phi,v,x_g,y_g,\phi_g,v_g)$ {
  if goToPose.`approachBallSuccess` $(x,y,\phi,v,x_g,y_g,\phi_g,v_g)$ **then**
    return goToPose.`executionDuration` $(x,y,\phi,v,x_g,y_g,\phi_g,v_g)$;
  else
    return INVALID;
  end
}

Algorithm 4: Modified `goToPose` action model for approaching the ball.

Analogously to Figure 11, the predicted execution durations of the two actions, as well as their summation are depicted in Figure 15. Invalid values are not rendered. The second graph depicts the function described in Algorithm 4. Note that due to removal of invalid values, the shape of the functions on the ground plane in the last two graphs corresponds to Figure 8 and 16(a).

In Figure 16(a), 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

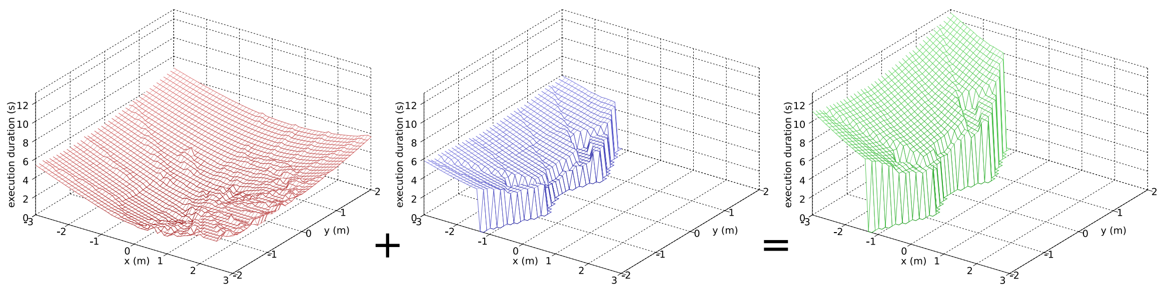
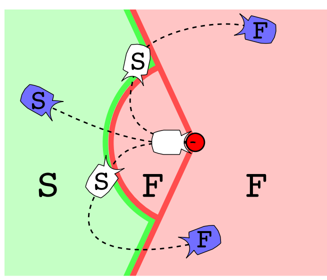


Figure 15: Search space for subgoal refinement in subgoal assertion.

when executing `goToPose`, and a subgoal is asserted. The subgoals, determined by subgoal refinement, are depicted as well.



(a) Three examples. Subgoal assertion is applied to two of them.

	Observed		
	`Fail`	`Success`	
Predicted			
`Fail`	2% $^{(52\%-50\%)}$	60% $^{(10\%+50\%)}$	→ 62%
`Success`	1%	37%	→ 38%
	↓ 3%	↓ 97%	

(b) Mean results over 100 episodes. Almost all predicted failures (50% of 52%) are transformed to yield a successful execution.

Figure 16: Results of subgoal assertion.

7.2 Empirical Evaluation

To evaluate automatic subgoal assertion, a hundred random ball approaches are executed in simulation, once with subgoal assertion, and once without. The results are summarized in Table 16(b). Without assertion, the results are similar to the results reported in Table 2. A collision is again correctly predicted approximately half the time: 52% of these hundred episodes. Subgoal assertion is applied in these cases, and is almost always successful: 50% of all episodes that are predicted to fail are now executed successfully. Only 2% of the episodes still have a collision, despite subgoal assertion. Because no subgoal assertion is applied when `Success` is predicted, there is no change in the lower prediction row. Consciously choosing not to apply subgoal assertion and not applying it are equivalent.

Subgoal assertion is applied unnecessarily in 10% of all episodes, which means that subgoal assertion is applied, even though the original sequence would already be successful. However, the execution with subgoal assertion and consequent subgoal refinement is a significant 8% slower than executing only the one `goToPose` action. The performance loss in these cases seems an acceptable cost compared to the pay-off of the dramatic increase in the number of successful task completions.

Summarizing: if subgoal assertion is not necessary, it is usually not applied. A subgoal is introduced half of the time, which raises successful task completion from 47 to 97%. Infrequently, subgoals are introduced inappropriately, which leads to a small loss of performance in terms of execution duration.

8. Conclusion and Outlook

Durative actions provide a conceptual abstraction that is reasoned about either by the designer during action selection design, or, if the abstraction is explicitly coded into the controller, by the action selection system itself. Action abstractions partially achieve their abstraction by ignoring action parameters. Although these parameters are not relevant to the action on an abstract level, they often are relevant to the performance and success of executing the plan.

If robots learn to predict the effects and performance of their actions, they can use this knowledge to optimize their behavior through subgoal refinement, and avoid failures through subgoal assertion. The empirical evaluations verify that this leads to more efficient, robust, and effective behavior. We believe these are important contributions towards bridging the gap between abstract planning systems and robot action execution.

In multi-robot scenarios such as robotic soccer, robots can be provided with the action models of teammates, enabling each robot to reason about the actions of others. We have shown that this enables robots to *implicitly* coordinate their actions with others, without having to resort to utility communication (Stulp et al., 2006). We have also successfully applied this approach to a heterogeneous team of robots, with robots from two different research groups (Isik, Stulp, Mayer, & Utz, 2006).

Also, we have preliminary results showing that even more accurate models can be learned when data is gathered online during operation time (Kirsch & Beetz, 2007). Extended operation times not only enable the robot to gather more training data, but actions are also called with parameterizations typical for the domain and the context in which the robot is used. This in contrast to our current method, which generates action parameterizations by randomly choosing them from *all possible* parameterizations. Given the same amount of data, a model that has to generalize over all possible parameterizations will tend to be less accurate than a model that has to do so over only a subset of parameterizations.

The main assumption underlying this article is that human-specified knowledge and robot-learned knowledge complement each other well in robot controllers. It is exemplary that recent winners of two well-known robotic benchmarks, the RoboCup mid-size league (Gabel, Hafner, Lange, Lauer, & Riedmiller, 2006) and the DARPA challenge (Thrun et al., 2006), emphasize that their success could only be achieved through the combination of manual coding and experience-based learning. More specifically, we believe that ideally human designers should only have to specify action abstractions offline. During task execution, the robot should then automatically optimize aspects of actions that are relevant for its execution with learned action models.

Future work includes learning several models for different performance measures, and optimizing several performance measures simultaneously. For instance, energy consumption is another important performance measure in autonomous mobile robots. By specifying objective functions that consist of the combinations of energy consumption and execution duration, they can both be optimized. By weighting individual performance functions differently, the function to be optimized can be customized to specific scenarios. For instance, in mid-size league robotic soccer, with its short operation time of 15 minutes, speed is far more important then energy consumption. In service robotics it is the other way around. Also, we intend to evaluate the use of other learning algorithms for predicting the failure of actions, for instance Neural Networks (Buck & Riedmiller, 2000) or Support Vector Machines (Hart, Ou, Sweeney, & Grupen, 2006). If these learning algorithms can predict action failures more accurately, even better results are to be expected from subgoal refinement and subgoal assertion.

Acknowledgments

We would like to thank Andreas Fedrizzi and the anonymous reviewers for their valuable remarks and suggestions. The research described in this article is partly funded by the German Research Foundation (DFG) in the SPP-1125, "Cooperating Teams of Mobile Robots in Dynamic Environments", and also by the CoTeSys cluster of excellence (Cognition for Technical Systems, `http://www.cotesys.org`), part of the Excellence Initiative of the DFG.

Appendix A

Robot	State Estimation		Belief State	Motor comm.
Soccer (real)	Probabilistic state estimation based on camera	$\Rightarrow$	$x, y, \phi, v,$	$v, \dot{\phi}$
	Probabilistic state estimation based on camera	$\Rightarrow$	x_{ball}, y_{ball}	
Soccer (simulated)	Ground truth, with noise based on noise model	$\Rightarrow$	$x, y, \phi, v,$	$v, \dot{\phi}$
	Ground truth if in field of view, with noise	$\Rightarrow$	x_{ball}, y_{ball}	
Kitchen	Ground truth	$\Rightarrow$	$x, y, \phi, v,$	$v, \dot{\phi}$
	Ground truth	$\Rightarrow$	ax, ay, az	ax, ay, az
	Ground truth if in field of view	$\Rightarrow$	$[x_o, y_o, z_o]^n$	
Arm control	Joint angles read directly from motor encoders	$\Rightarrow$	$\theta^a, \dot{\theta}^a, \theta^b, \dot{\theta}^b$	I^a, I^b

Table 5: State variables in the belief state, state estimation process used to acquire them, and the motor command in each domain. x, y, ϕ, v is dynamic pose of robot body, ax, ay, az is relative position of arm to robot body, $[x_o, y_o, z_o]^n$ are absolute positions of n objects, $v, \dot{\phi}$ is translational/rotational velocities, I is current sent to arm joint motor.

Robot	Action	Action Parameters: Current	Action Parameters: Goal
Soccer	`goToPose`	x, y, ϕ, v	x_g, y_g, ϕ_g, v_g
	`dribbleBall`	$x, y, \phi, v, x_{ball}, y_{ball}$	x_g, y_g, ϕ_g, v_g
Kitchen	`goToPose`	x, y, ϕ, v	x_g, y_g, ϕ_g, v_g
	`reach`	x, y, z, ax, ay, az	$x_g, y_g, z_g, ax_g, ay_g, az_g$
Arm control	`reach`	$\theta^a, \dot{\theta^a}, \theta^b, \dot{\theta^b}$	$\theta^a_g, \dot{\theta}^a_g, \theta^b_g, \dot{\theta}^b_g$

Table 6: List of actions used in the application domains. The list of actions might be shorter than expected. For instance, it is doubtful that robots could play soccer if they can only navigate to a certain pose. It is the aim of this article to demonstrate how only a few actions can be reused and customized to perform well in varying task contexts.

Robot	Action	Features
Soccer	`goToPose` and `dribbleBall`	$v, dist = \sqrt{(x - x_g)^2 + (y - y_g)^2}$, $angle\_to = \lvert angle\_to_{signed} \rvert$, $angle\_at = sgn(angle\_to_{signed}) \cdot norm(\phi_g - atan2(y_g - y, x_g - x))$
Kitchen	`goToPose`	$v_g, dist, angle\_to, angle\_at$, $\Delta angle = \lvert norm(\phi_g - \phi) \rvert$
	`reach`	$dist_{xyz} = \sqrt{(x - x_g)^2 + (y - y_g)^2 + (z - z_g)^2}$ $dist_{xy} = \sqrt{(x - x_g)^2 + (y - y_g)^2}, dist_{xz}, dist_{yz}$, $angle_{xy} = atan2(y_g - y, x_g - x), angle_{xz}, angle_{yz}$
Arm control	`reach`	$dist = \sqrt{(\theta^a - \theta^a_g)^2 + (\theta^b - \theta^b_g)^2}$, $angle_1 = norm(atan2(\theta^b_g - \theta^b, \theta^a_g - \theta^a) - atan2(\dot{\theta^b}, \dot{\theta^a}))$, $angle_2 = norm(-atan2(\theta^b_g - \theta^b, \theta^a_g - \theta^a) + atan2(\dot{\theta}^b_g, \dot{\theta}^a_g))$ $v = \sqrt{\dot{\theta^a}^2 + \dot{\theta^b}^2}$, $v_g = \sqrt{\dot{\theta}^{a\,2}_g + \dot{\theta}^{b\,2}_g}$

$norm(a)$: adds or subtracts 2π to a until is in range $[-\pi, \pi]$
$angle\_to_{signed} = norm(atan2(y_g - y, x_g - x) - \phi)$

Table 7: The feature spaces used to learn action models

References

Arkin, R. (1998). *Behavior based Robotics*. MIT Press, Cambridge, Ma.

Åström, K. J., & Murray, R. M. (2008). *Feedback Systems: An Introduction for Scientists and Engineers*, chap. Supplement on optimization-based control. Princeton University Press.

Barto, A., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete event systems*, *13*(1-2), 41–77.

Beetz, M., & Belker, T. (2000). XFRMLearn - a system for learning structured reactive navigation plans. In *Proceedings of the 8th International Symposium on Intelligent Robotic Systems*.

Beetz, M. (2000). *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, Vol. LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers.

Beetz, M. (2001). Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99*, *4*, 25–55.

Beetz, M., Schmitt, T., Hanek, R., Buck, S., Stulp, F., Schröter, D., & Radig, B. (2004). The AGILO robot soccer team – experience-based learning and probabilistic reasoning in autonomous robot control. *Autonomous Robots*, *17*(1), 55–77.

Belker, T. (2004). *Plan Projection, Execution, and Learning for Mobile Robot Control*. Ph.D. thesis, Department of Applied Computer Science, University of Bonn.

Bellingham, J., Richards, A., & How, J. P. (2002). Receding horizon control of autonomous aerial vehicles. In *Proceedings of the 2002 American Control Conference*, Vol. 5, pp. 3741–3746.

Benson, S. (1995). Inductive learning of reactive action models. In *International Conference on Machine Learning (ICML)*, pp. 47–54.

Bertram, D., Kuffner, J., Dillmann, R., & Asfour, T. (2006). An integrated approach to inverse kinematics and path planning for redundant manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1874–1879.

Bouguerra, A., & Karlsson, L. (2005). Symbolic probabilistic-conditional plans execution by a mobile robot. In *IJCAI-05 Workshop: Reasoning with Uncertainty in Robotics (RUR-05)*.

Brock, O., & Khatib, O. (1999). Elastic Strips: A framework for integrated planning and execution. In *Proceedings of the International Symposium on Experimental Robotics*, pp. 329–338.

Buck, S., Beetz, M., & Schmitt, T. (2002a). M-ROSE: A Multi Robot Simulation Environment for Learning Cooperative Behavior. In Asama, H., Arai, T., Fukuda, T., & Hasegawa, T. (Eds.), *Distributed Autonomous Robotic Systems 5, Lecture Notes in Artificial Intelligence*, LNAI. Springer-Verlag.

Buck, S., Beetz, M., & Schmitt, T. (2002b). Reliable Multi Robot Coordination Using Minimal Communication and Neural Prediction. In Beetz, M., Hertzberg, J., Ghallab, M., & Pollack, M. (Eds.), *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.

Buck, S., & Riedmiller, M. (2000). Learning situation dependent success rates of actions in a RoboCup scenario. In *Pacific Rim International Conference on Artificial Intelligence*, p. 809.

Cambon, S., Gravot, F., & Alami, R. (2004). A robot task planner that merges symbolic and geometric reasoning.. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pp. 895–899.

Clement, B. J., Durfee, E. H., & Barrett, A. C. (2007). Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research*, *28*, 453–515.

Coradeschi, S., & Saffiotti, A. (2001). Perceptual anchoring of symbols for action. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 407–416.

Dean, T., & Wellmann, M. (1991). *Planning and Control*. Morgan Kaufmann Publishers.

Dearden, A., & Demiris, Y. (2005). Learning forward models for robotics. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1440–1445.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Fox, M., & Long, D. (2003). PDDL2.1: An extension of PDDL for expressing temporal planning domains.. *Journal of Artificial Intelligence Research*, *20*, 61–124.

Fox, M., Ghallab, M., Infantes, G., & Long, D. (2006). Robot introspection through learned hidden markov models. *Artificial Intelligence*, *170*(2), 59–113.

Gabel, T., Hafner, R., Lange, S., Lauer, M., & Riedmiller, M. (2006). Bridging the gap: Learning in the RoboCup simulation and midsize league. In *Proceedings of the 7th Portuguese Conference on Automatic Control*.

Gerkey, B., Vaughan, R. T., & Howard, A. (2003). The Player/Stage Project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR)*, pp. 317–323.

Ginsberg, M. L. (1989). Universal planning: an (almost) universally bad idea. *AI Magazine*, *10*(4), 40–44.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Kluwer Academic Publishers.

Haigh, K. Z. (1998). *Situation-Dependent Learning for Interleaved Planning and Robot Execution*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Harris, C. M., & Wolpert, D. M. (1998). Signal-dependent noise determines motor planning. *Nature*, *394*(20), 780–784.

Hart, S., Ou, S., Sweeney, J., & Grupen, R. (2006). A framework for learning declarative structure. In *Workshop on Manipulation for Human Environments, Robotics: Science and Systems*.

Haruno, M., Wolpert, D. M., & Kawato, M. (2001). MOSAIC model for sensorimotor learning and control. *Neural Computation*, *13*, 2201–2220.

Haruno, M., Wolpert, D. M., & Kawato, M. (1999). Multiple paired forward-inverse models for human motor learning and control. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pp. 31–37, Cambridge, MA, USA. MIT Press.

Hoffmann, J., & Düffert, U. (2004). Frequency space representation and transitions of quadruped robot gaits. In *Proceedings of the 27th Australasian computer science conference (ACSC)*, pp. 275–278.

Infantes, G., Ingrand, F., & Ghallab, M. (2006). Learning behavior models for robot execution control. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*, pp. 678–682.

Isik, M., Stulp, F., Mayer, G., & Utz, H. (2006). Coordination without negotiation in teams of heterogeneous robots. In *Proceedings of the RoboCup Symposium*, pp. 355–362.

Jacobs, R. A., & Jordan, M. I. (1993). Learning piecewise control strategies in a modular neural network. *IEEE Transactions on Systems, Man and Cybernetics*, *23*(3), 337–345.

Jaeger, H., & Christaller, T. (1998). Dual dynamics: Designing behavior systems for autonomous robots. *Artificial Life and Robotics*, *2*(3), 108–112.

Jordan, M. I., & Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science*, *16*, 307–354.

Kirsch, A., & Beetz, M. (2007). Training on the job — collecting experience with hierarchical hybrid automata. In Hertzberg, J., Beetz, M., & Englert, R. (Eds.), *Proceedings of the 30th German Conference on Artificial Intelligence (KI-2007)*, pp. 473–476.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., & Osawa, E. (1997). RoboCup: The robot world cup initiative. In *Proceedings of the first international conference on autonomous agents (AGENTS)*, pp. 340–347.

Kollar, T., & Roy, N. (2006). Using reinforcement learning to improve exploration trajectories for error minimization. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pp. 3338–3343.

Koska, W. (2006). Optimizing autonomous service robot plans by tuning unbound action parameters. Master's thesis, Technische Universiät München.

Kovar, L., & Gleicher, M. (2003). Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on computer animation (SCA)*, pp. 214–224.

Müller, A., & Beetz, M. (2006). Designing and implementing a plan library for a simulated household robot. In Beetz, M., Rajan, K., Thielscher, M., & Rusu, R. B. (Eds.), *Cognitive Robotics: Papers from the AAAI Workshop*, Technical Report WS-06-03, pp. 119–128, Menlo Park, California. American Association for Artificial Intelligence.

Nakanishi, J., Cory, R., Mistry, M., Peters, J., & Schaal, S. (2005). Comparative experiments on task space control with redundancy resolution. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pp. 3901–3908.

Nilsson, N. J. (1984). Shakey the robot. Tech. rep. 323, AI Center, SRI International.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

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

Quinlan, R. (1992). Learning with continuous classes. In Adams, A., & Sterling, L. (Eds.), *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence - A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey.

Ryan, M., & Pendrith, M. (1998). RL-TOPs: an architecture for modularity and re-use in reinforcement learning. In *Proceedings 15th International Conference on Machine Learning (ICML)*, pp. 481–487.

Saffiotti, A., Ruspini, E. H., & Konolige, K. (1993). Blending reactivity and goal-directedness in a fuzzy controller. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pp. 134–139, San Francisco, California. IEEE Press.

Schaal, S., & Schweighofer, N. (2005). Computational motor control in humans and robots. *Current Opinion in Neurobiology*, *15*, 675–682.

Schmill, M. D., Oates, T., & Cohen, P. R. (2000). Learning planning operators in real-world, partially observable environments. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (ICAPS)*, pp. 246–253.

Shahaf, D., & Amir, E. (2006). Learning partially observable action schemas.. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*.

Simmons, G., & Demiris, Y. (2004). Biologically inspired optimal robot arm control with signal-dependent noise. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems (IROS)*, pp. 491–496.

Smith, D., Frank, J., & Jónsson, A. (2000). Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, *15*(1), 47–83.

Stulp, F. (2007). *Tailoring Robot Actions to Task Contexts using Action Models*. Ph.D. thesis, Technische Universität München.

Stulp, F., & Beetz, M. (2005). Optimized execution of action chains using learned performance models of abstract actions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*.

Stulp, F., Isik, M., & Beetz, M. (2006). Implicit coordination in robotic teams using learned prediction models. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1330–1335.

Stulp, F., Koska, W., Maldonado, A., & Beetz, M. (2007). Seamless execution of action sequences. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3687–3692.

Sussman, G. J. (1973). *A computational model of skill acquisition.* Ph.D. thesis, Massachusetts Institute of Technology.

Thrun, S. et al. (2006). Stanley, the robot that won tFhe DARPA grand challenge. *Journal of Field Robotics*, *23*(9), 661–692.

Todorov, E., Li, W., & Pan, X. (2005). From task parameters to motor synergies: A hierarchical framework for approximately optimal control of redundant manipulators. *Journal of Robotic Systems*, *22*(11), 691–710.

Uno, Y., Wolpert, D. M., Kawato, M., & Suzuki, R. (1989). Formation and control of optimal trajectory in human multijoint arm movement - minimum torque-change model. *Biological Cybernetics*, *61*(2), 89–101.

Utz, H., Kraetzschmar, G., Mayer, G., & Palm, G. (2005). Hierarchical behavior organization. In *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS)*, pp. 2598–2605.

Wimmer, M., Stulp, F., Pietzsch, S., & Radig, B. (2008). Learning local objective functions for robust face model fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, *30*(8). to appear.

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques* (2^{nd} edition). Morgan Kaufmann, San Francisco.

Wolpert, D., & Ghahramani, Z. (2000). Computational principles of movement neuroscience. *Nature Neuroscience Supplement*, *3*, 1212–1217.

Wolpert, D. M., & Flanagan, J. (2001). Motor prediction. *Current Biology*, *11*(18), 729–732.

Younes, H. L. S., & Simmons, R. G. (2003). VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, *20*, 405–430.