Institut für Informatik
der Technischen Universität München

Tailoring Robot Actions to Novel Task Contexts using Action Models

Dissertation

Freek Stulp

Institut für Informatik

der Technischen Universität München

Tailoring Robot Actions to Novel Task Contexts using Action Models

Freek Stulp

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		....
Prüfer der Dissertation:		
	1.	....
	2.	....
	3.	....

Die Dissertation wurde am .... bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am .... angenommen.

Abstract

In motor control, high-level goals must be expressed in terms of low-level motor commands. An effective method to bridge this gap, widespread in both nature and robotics, is to acquire a set of temporally extended actions, each designed for specific goals and task contexts. An action selection module then selects the appropriate action in a given situation. In this approach, high-level goals are mapped to actions, and actions produce streams of motor commands. The first mapping is often ambiguous, as several actions or action parameterizations can achieve the same goal. Instead of choosing an arbitrary action or parameterization, the robot should select those that best fulfill some pre-specified requirement, such as minimal execution duration, successful execution, or coordination of actions with others.

The key to being able to perform this selection lies in prediction. By predicting the performance of different actions and action parameterizations, the robot can also predict which of them best meets the requirement. Action models, which have many similarities with human forward models, enable robots to make such predictions.

In this dissertation, we will introduce a computational model for the acquisition and application of action models. First, action models are learned from observed experience. Three applications of action models will be presented. 1) *Subgoal refinement*, which enables robots to optimize actions in action sequences by predicting which action parameterization leads to the best performance. 2) *Condition refinement* and *subgoal assertion*, with which robots can adapt existing actions to novel task contexts and goals by predicting when action execution will fail. 3) *Implicit coordination*, that multiple robots can use to coordinate their actions, by locally making predictions about the performance of other robots. The acquisition and applications of action models have been evaluated in three robotic domains: the Pioneer I robots of our RoboCup mid-size league team, a simulated B21 in a kitchen environment, and a PowerCube robotic arm.

The main principle behind this approach is that in robot controller design, knowledge that robots learn from observed experience themselves complements well the abstract knowledge that humans specify.

Zusammenfassung

In der Bewegungssteuerung müssen abstrakte Ziele in konkreten Bewegungsbefehlen ausgedrückt werden. In der Natur wie in der Robotik kann diese Kluft durch Aktionen überwunden werden, die für spezifische Ziele und Aufgabenkontexte bestimmt sind. Ein spezielles Modul wählt dann die Aktionen aus, welche sich für die jeweilige Situation eignen. Die Abbildung von Zielen auf Aktionen ist häufig vieldeutig, da mehrere Aktionen oder Aktionsparametrisierungen das gleiche Ziel erreichen können. Statt eine beliebige Aktion oder Aktionsparametrisierung zu wählen, sollte der Roboter jene bevorzugen, die eine vordefinierte Anforderung erfüllen, wie etwa minimale Ausführungsdauer, Ausführungserfolg oder Koordination mit anderen Robotern.

Die Vorhersage der Leistung bestimmter Aktionen erlaubt es dem Roboter zu erkennen, welche Aktion oder Aktionsparametrisierung die Anforderung am Besten erfüllen werden. Aktionsmodelle, die Ähnlichkeit mit dem ‘Forward Model’ des Menschen haben, ermöglichen Robotern, solche Vorhersagen zu machen.

In dieser Dissertation stellen wir ein Berechnungsmodell für den Erwerb und die Anwendung dieser Aktionsmodelle vor. Zuerst werden Aktionsmodelle aus beobachteter Erfahrung erlernt. Drei Anwendungen der Aktionsmodelle werden dargestellt. 1) *Subgoal Refinement*, das Aktionen in den Aktionsketten optimiert, indem es voraussagt, welche Aktionsparametrisierung zur besten Leistung führen wird. 2) *Condition Refinement* und *Subgoal Assertion*, die vorhandene Aktionen neuen Aufgabenkontexten und Zielen anpassen, indem sie voraussagen, wann die Aktionsdurchführung fehlschlagen wird. 3) *Implicit Coordination*, mit deren Hilfe Roboter durch lokale Vorhersagen über die Leistung anderer Roboter ihre Aktionen koordinieren können. Der Erwerb und die Anwendungen der Aktionsmodelle sind ausgewertet worden auf Pioneer I Fussballrobotern, auf einem simulierten B21 in einer Küchenumgebung, und bei der Steuerung eines PowerCube Arms.

Das Hauptprinzip dieses Ansatzes besteht darin, dass beim Entwurf von Robotersteuereinheiten das Wissen, das sich Roboter selbst durch Beobachtung aneignen, jenes durch den Menschen bestimmte abstrakte Wissen gut komplettiert.

Contents

List of Figures

List of Tables

1. Introduction

"It is the ability to make predictions about the future that is the crux of intelligence."

Jeff Hawkins

In motor control, there is a distinction between knowing *what* to do and knowing *how* to do it. This distinction is apparent in the human brain, where declarative and procedural knowledge is acquired, stored and accessed in different ways (Scoville and Milner, 1957; Cavaco et al., 2004). Let us illustrate the difference between declarative and procedural knowledge with the soccer scenario in Figure 1.1, in which the goal is to be in possession of the ball in front of the goal. *What* needs to be done to achieve this goal can be informally declared as: "First approach the ball, and then dribble it towards the opponent goal."

In both nature (Wolpert and Ghahramani, 2000; Baerends, 1970) and robotics (Arkin, 1998), such abstract plans are often mapped to actions. Actions are temporally extended control routines that achieve specific goals, and only apply to certain task contexts. In the example, the declarative knowledge could be mapped to the actions `approachBall` and `dribbleBall`. With these actions, the robot now also knows *how* to achieve its goal[1].

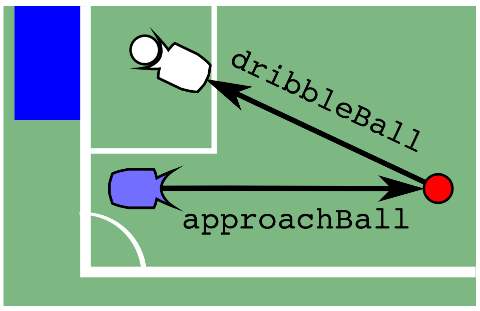
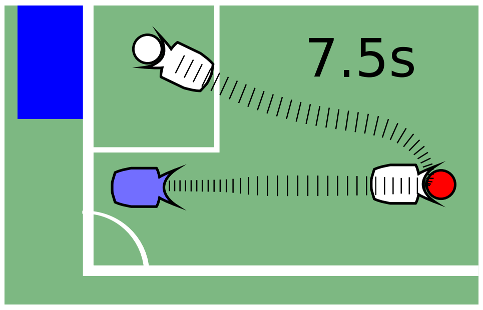


Figure 1.1. Soccer scenario

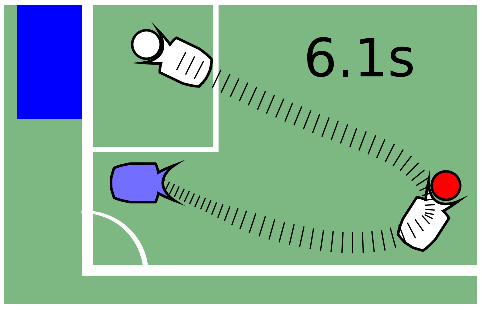
However, a problem remains. Although the actions specify how to achieve the goal, there are often several ways to execute them. Figure 1.2 depicts two executions of the same action sequence. In the first, the robot naively executes the first action, and arrives at the ball with the goal at its back, as depicted in Figure 1.2(a). This is an unfortunate position from which to start dribbling towards the goal. An abrupt transition occurs between the actions, as the robot

[1]Note that we interpret the terms 'procedural' and 'declarative' as they are used in cognitive science (Cavaco et al., 2004), not as in the debate on logic knowledge representation in the 60s and 70s (Winograd, 1975).

needs to brake to slowly and carefully maneuver itself behind the ball in the direction of the goal.



(a) An execution with an abrupt transition at the intermediate goal.



(b) A time-optimal execution that exhibits smooth motion.

Figure 1.2. Two alternative executions of the same action sequence.

Preferably, the robot should go to the ball *in order* to dribble it towards the goal afterwards. The robot should, as depicted in the Figure 1.2(b), perform the first action sub-optimally in order to achieve a much better position for executing the second action. The behavior shown in Figure 1.2(b) has a higher performance, achieving the ultimate goal in less time.

This example demonstrates that although the angle of approach might not be relevant on an abstract level, it does influence execution performance. But what exactly is the best angle of approach? Unfortunately, neither declarative nor procedural knowledge suffices to answer this question. This is the remaining problem referred to previously.

In this dissertation, we will demonstrate that the key to solving this problem lies in a third kind of 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 prediction models through experimentation, observation and generalization. It does so by simply recording the results of executing the action with different parameterizations, and training learning algorithms with the data so acquired.

1.1 Key Principles

One of the main motivations behind robotics research is to develop robots that can assist with or assume tasks that are either too dangerous or too tedious for humans. Prolonging and increasing the independence of the disabled and the elderly with assisting technologies such as robots is also predicted to have a large social impact (Cortés et al., 2003). Examples of such tasks are performing rescue operations, autonomous driving, providing mobility for the disabled, and doing the dishes.

Although there are several projects and conferences committed to robots that learn more or less from scratch how to act in the real-world (Metta et al., 2006; Kaplan et al., 2006), the resulting robots have certainly not yet reached a level where they can perform the tasks described above. Currently, systems designed to achieve such real-world tasks still require the designers to encode their knowledge about how to solve real-world problems into the robot controller. For instance, action selection modules are still often manually encoded as state-machines (Lötzsch et al., 2004; Obst, 2002; Murray, 2001). Here, the designer directly encodes knowledge about which functional state the robot is in, and which action should be executed in this state.

However, through experimentation, observation and generalization, robots can learn complementary knowledge, and use it to improve, adapt and optimize their controllers. Learned knowledge can often be used to make decisions that are difficult for humans to make. Furthermore, experience-based learning is grounded in real world observations, not human intuition. Prediction models are a good example of knowledge that can often be learned more accurately and efficiently than that they can be specified manually.

It is exemplary that the 2006 winners of two well-known robotic benchmarks, the RoboCup mid-size league (Gabel et al., 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.

The main principle in this dissertation is therefore that *human-specified knowledge and robot-learned knowledge complement each other well in robot controllers*. The introduction and example in Figure 1.1 have briefly illustrated the other key principles on which this dissertation is based:

Principle I Declarative knowledge can be explicitly specified by humans.

Principle II Procedural knowledge can be segmented into durative actions.

Principle III Mapping declarative to procedural knowledge is ambiguous, and choosing the mapping affects performance and behaviour.

Principle IV This ambiguity can be resolved with predictive knowledge, which leads to more effective and efficient action execution.

Principle V Predictive knowledge can be learned from observed experience

These key principles will be discussed in more detail in the sections referred to by the numbers in the list. We will explain that the first two principles are well established in robotics and cognitive science. The third principle is essentially the problem statement, and the last two are the solution ideas.

Principle I Declarative knowledge: human specified

An important aspect of declarative knowledge is that it is consciously accessible, and allows us to declare our intentions and plans to others. An example was given in Figure 1.1, in which the task could be informally declared as: "First approach the ball, and then dribble it towards the goal." Other examples from soccer are: "Approaching the ball is much like navigating, except that you should not bump into the ball before the desired pose at the ball is achieved." or "To regain ball possession, only one player should approach the ball."

These statements are at a level of abstraction that makes them valid for both human and robot soccer players. The validity in both domains enables the transfer of declarative knowledge from humans to robots, and programmers usually have no problem in encoding this knowledge in the controller. It also enables humans to give advice to robots in a declarative way (Carpenter et al., 2002).

In planning approaches, this is explicitly done with declarative languages such as PDDL (Fox and Long, 2003). The knowledge can also be implicitly encoded using the control flow of the programming language. However, with the latter the robot cannot reason about or manipulate this knowledge, and the encoding can be such that even other designers cannot recognize the intentions from the code.

For now, it is not so important how declarative knowledge is represented in the controller, as long as it is clear that at some point during controller design, a designer will have explicitly thought about the declarative statements above, and coded them in the controller's language. Examples of both explicitly and implicitly representing declarative knowledge in robot controllers will be given in Sections 5.2.1 and 5.2.3 respectively.

Principle II Procedural knowledge: durative actions

The famous patient H.M. provided the first proof for the difference between declarative and procedural memory storage (Scoville and Milner, 1957). At the age of 27, a bilateral medial temporal lobe resection was carried out to to correct his increasingly debilitating epilepsy. During the operation, the amygdala, uncus, hippocampal gyrus, and anterior two-thirds of the hippocampus were removed. After the operation, H.M. was incapable of storing any novel declarative facts, although the facts before his operation were retained. On the other hand, H.M. could learn novel skills, such as mirror-tracing (Gabrieli et al., 2004). Surprisingly, H.M. improves at mirror-tracing tasks over time with training, but has no recollection of ever having done this task before.

As H.M. demonstrates, procedural knowledge is not explicitly and consciously accessible to humans, in contrast to declarative knowledge. This is probably the reason why programmers find it more difficult to transfer procedural knowledge to robots. Also, although abstract descriptions of tasks are valid in general, procedural knowledge is often very platform-dependent. For instance, there might be differences in locomotion (biped vs. wheeled), controllable degrees of freedom (non-holonomic vs. holonomic), and motor commands (action potentials vs. voltages).

Wolpert and Ghahramani (2000) describe well the difficulty of mapping declarative knowledge to procedural knowledge in the human motor system: "Everyday tasks are generally specified at a high, often symbolic level, such as taking a drink of water from a glass. However, the motor system must eventually work at a detailed level, specifying muscle activations leading to joint rotations and the path of the hand in space. There is clearly a gap between the high-level task and low-level control.".

Using durative actions to bridge this gap has proven to be a successful approach in both nature (Baerends, 1970; Wolpert and Ghahramani, 2000) and robotics (Arkin, 1998). Actions encapsulate knowledge about how certain goals can be achieved in certain task contexts. 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 achieve different goals within different soccer contexts. Because actions only apply to limited task contexts, they are easier to design or learn than a controller that must be able to deal with all possible contexts (Haruno et al., 1999; Jacobs and Jordan, 1993). In cognitive science, actions are known as *inverse models*, and in robotics as *behaviors*, *routines*, or, confusingly, *controllers*. In Table 3.1, we will list which specific research field uses which terminology.

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. 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 benefits: 1) The controller is less complex, making it more robust. 2) Fewer interactions between action need to be considered, which facilitates 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.

To achieve more complex tasks, actions can be combined and concatenated, using declarative knowledge. As we saw in the example, “First approach the ball, and then dribble it towards the goal.” can be mapped to the action sequence `approachBall`, `dribbleBall`.

So, declarative knowledge maps goals to actions, and procedural knowledge maps actions to motor commands, which can be directly applied to the motor system. This divide and conquer approach to control helps to bridge the gap between high-level goals and low-level motor commands.

Principle III Mapping is Ambiguous

Mapping goals to actions is often ambiguous due to redundancy of actions, so several actions or several action parameterizations can achieve the same goal. This is a well known principle in human motor control, where there are often more degrees of freedom available than are strictly needed to solve a task (Schaal and Schweighofer, 2005). Actions are then said to be redundant or overexpressive, and the freedom of movement that is not constrained by the task is called the uncontrolled manifold in cognitive science (Scholz and Schöner, 1999), and null-space in engineering (Hooper, 1994; Nakanishi et al., 2005). The redundancy of actions raises an important question. How should the excess degrees of freedom be parameterized? This problem is known as the degree-of-freedom problem, or problem of redundancy resolution (Schaal and Schweighofer, 2005).

In the running example in Figure 1.2(a) for instance, we saw that the action sequence that arises from the declarative knowledge can actually be executed in many ways. This example illustrates that some questions still remain. “First approach the ball, and then dribble it towards the goal.” maps to the action sequence `approachBall`, `dribbleBall`. But what is the best angle of approach? From an abstract point of view, being at the ball is sufficient for dribbling it. Although the angle of approach might not be relevant to the task on an abstract level, the example clearly shows that it does influence execution performance.

The same holds for the other statements: “To regain ball possession, only one player should approach the ball.” But which player should this be? Probably the fastest. But exactly who is the fastest? “Approaching the ball is much like navigating, except that you should not bump into the ball before the desired pose at the ball is achieved.” But exactly when does the robot bump into the ball?

One of the benefits of actions is that they can be designed or learned independently of other actions. The questions arise when actions are executed in contexts for which they were not initially designed. For instance, “Which angle of approach is the best?” arose from executing the action in the context of action sequences, and “When will the robot bump into the ball?” arose from navigating in the context of approaching the ball. Finally, “Who will be the fastest?” arose from the context of playing in a multi-robot team.

One way to answer these questions is to design or learn new actions that are tailored to the novel context in which the question arose. Instead of using the general `approachBall(Pose)` in the scenario in Figure 1.2, a new action `approachBallInOrderToDribbleBall(Position,Pose)` is designed or learned. This customized action takes into account that the robot should dribble the ball to a certain position afterwards. It therefore takes the next location as a parameter, and the action internally computes the optimal angle of approach. Although this customized action might perform better and yield smoother motion in this context, its long name already clearly implies the loss of generality. This manual action customization soon becomes a laborious task, as each task context, and there are usually many, would require their own task-specific action. In the next section, we present an alternative solution, which reuses existing actions based on predictive knowledge, and motivate why it is preferable to designing or learning novel actions.

Principle IV Predictive knowledge enables effective control

Although the actions in this dissertation themselves are fixed, this does not mean that their application is fixed. Much freedom remains in the way actions are parameterized, and also in which actions are executed in the first place. For instance, the original `approachBall(Pose)` can be used very well to achieve the optimal execution in Figure 1.2(b), if its parameter determining the angle of approach is correctly set.

Here, the benefit of having action parameters becomes clear. The `approachBallInOrderToDribbleBall` action does not have the angle of approach as an action parameter, but somehow computes an optimal angle ‘inside’ the action itself. However, which angle is optimal depends on what is being optimized: time, energy consumption, traveled distance, etc. It also depends on which action will follow: a fast dribble to score, a careful dribble to prepare for passing the ball, etc. To achieve good performance, each of these contexts would require its own customized action. Instead, it is better to have the angle of approach in the parameter list of a more general action `approachBall`, which can achieve all these tasks. Exactly which angle of approach is best in the current task context can be determined on-line ‘outside’ of the action. With this approach, ***existing* actions can be tailored to novel task contexts**. Adapting or refining already existing actions so that they can solve novel tasks alleviates the need to design or learn new actions. This leads to fewer actions, with all the benefits previously discussed.

By implementing the novel action `approachBallInOrderToDribbleBall`, the designer is specifying *how* an action can be executed best in the context of action sequences. Again, this is tedious and error-prone. It would be more convenient if the designer would only have to declare requirements that action execution should meet, such as “Execute action sequences as quickly as possible.”, or “Do not bump into the ball when approaching it.”. Given the freedom caused by the redundancy of actions, the robot should then attempt to fulfill these requirements by tailoring actions on-line. In the running example for instance, the robot is required to minimize the expected execution duration of the overall action sequence. (Schaal and Schweighofer, 2005) calls these requirements ‘subordinate criteria’, and (Wolpert and Ghahramani, 2000) refers to them as ‘cost functions’.

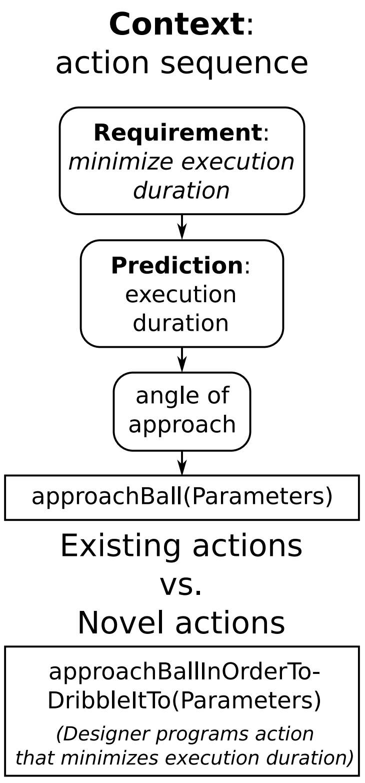

Context: action sequence
Requirement: minimize execution duration
Prediction: execution duration
angle of approach
approachBall(Parameters)
Existing actions vs. Novel actions
approachBallInOrderTo-DribbleItTo(Parameters)
(Designer programs action that minimizes execution duration)

Figure 1.3. Existing actions vs. Novel actions

Note that these requirements are independent of the action implementation, and hold for a variety of actions and task contexts, which makes them generally applicable, and therefore easy to formulate. On the other hand, the parameters and actions that fulfill these requirements depend very strongly on action implementations and task contexts, and will be different for each of them. Therefore, the robot should preferably determine these parameters autonomously on-line. This approach enables the designer to **specify requirements, rather than novel actions**.

Transforming actions or choosing action parameterizations to fulfill requirements is only possible if the robot can predict the outcome of actions and their parameterizations. Fulfilling the requirement “Execute action sequences as quickly as possible.” can only be done if the robot knows which action sequence will the fastest beforehand. The requirement “Do not bump into the ball when approaching it.” can only be fulfilled if the robot can predict if it will bump into the ball in some situation. Knowing which robot is the quickest to the ball is only possible if each robot can predict the approach time to the ball for each robot. Being able to predict the consequences of actions is essential to answering the questions that arise from Principle III, and **robots can tailor existing actions *themselves* with predictive knowledge**.

This approach has been informally depicted in Figure 1.3. The first step in reusing actions is to specify a requirement. Then, the predictions relevant to fulfill this requirement are made.

This yields an action selection or action parameters. The execution is then performed with existing actions. Note that these three steps have been printed in bold in the previous paragraphs. The questions related to efficient and effective execution of actions are so kept outside of the action.

On the other hand, when designing novel actions for novel task contexts, the designer contemplates the requirements, makes predictions her/himself, and implicitly codes them in the new action, as depicted in Figure 1.3. In our approach, no new actions are created, but existing actions are reused, refined and tailored to novel task contexts. With predictive knowledge, robots can tailor actions to novel task contexts themselves. This alleviates the need for designers to adapt or refine actions manually, and makes the robot more autonomous.

Principle V Predictive knowledge can be learned

Action models enable robots to predict the performance or outcome of actions, given a certain parameterization. Examples are predicting the expected execution duration, or whether a action is likely to succeed. But how is this predictive knowledge acquired?

It is learned from observed experience. First, each action is executed for a multitude of parameterizations and the performances and outcomes are recorded. A learning algorithm then learns a generalized model that maps an action and its parameterization to expected performance. In the soccer domain for instance, robots learn to predict the execution duration of their `goToPose` action by simply navigating to random locations on the field and recording the duration. Model trees are then trained with this data, after having transformed it to an appropriate feature space,

The advantage of this approach over analytical methods is that it is based on real experience, and therefore takes all factors relevant to performance into account. Also, many hand-coded actions are difficult to formalize analytically, or analysis is impossible because the inner workings of the action are unknown. In principle, learning models can also be done on-line, so that action models can adapt to changing environments (Dearden and Demiris, 2005).

Beetz and Belker (2000) summarize well the difficulty of analytically specifying action models for navigation actions: “Navigation behavior is the result of the subtle interplay of many complex factors. These factors include the robot’s dynamics, sensing capabilities, surroundings, parameterizations of the control program, etc. It is impossible to provide the robot with a deep model for diagnosing navigation behavior.”

Summary

In the previous, we have deduced the solution idea from what the designer can conveniently specify, and what the robot can learn from experience. This train of reasoning can be inductively reversed and summarized as follows:

- Although actions are immutable (in this dissertation), there is still freedom in how they are parameterized and in which contexts they are executed.
- This freedom allows actions to be tailored to novel contexts.
- Predictive knowledge, which the robot can learn from observed experience, enables the robot to tailor actions itself.
- Off-line, the designer can specify requirements that action execution should meet, which the robot takes into account when tailoring actions on-line.
- This is preferable to designing novel customized actions, as requirements are more general, and fewer actions lead to more adaptive and robust controllers.

At his point, we would like to draw attention to the role of cognitive science in this dissertation. There is an increasing interest in exploiting human strategies for dealing with complex control in robotics (Lopes and Santos-Victor, 2005; Sloman, 2006; Dearden and Demiris, 2005), and there is an increasing exchange between terminologies and formalizations used in cognitive science and robotics. Action models, which are inspired by forward models found in humans, are a good example of this exchange. Throughout the dissertation, we will therefore also discuss cognitive science research that focuses on the acquisition and application of predictive models. Although this research is an important source of inspiration, in this dissertation the goal is not to explicitly model cognitive processes, or to reproduce empirical results from cognitive science.

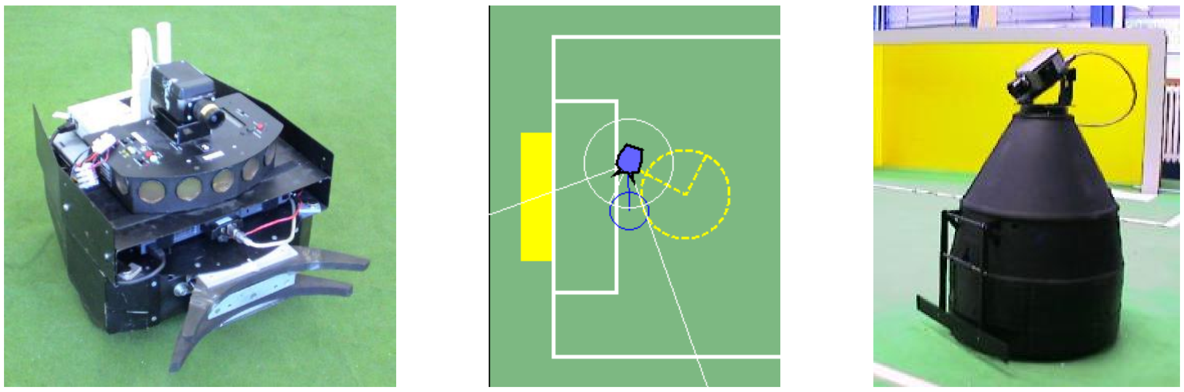
1.2 Robotic Domains

The key principles have been implemented in and applied to three robotic domains: robotic soccer, service robotics and arm control. Such a variety of robots and domains has been chosen to emphasize the generality of the system. Also, the different characteristics of the domains allow different aspects of action model applications to be investigated.

1.2.1 Robotic soccer

RoboCup is an international joint project to promote AI, robotics, and related fields. It is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined. The central topic of research is the soccer game, aiming at innovations that can be applied to socially and industrially significant problems. The ultimate goal of the RoboCup project is that by mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA, against the winner of the most recent World Cup (Kitano et al., 1997).

Within RoboCup, there are several leagues, each with their own technological and research challenges. The team of the Technische Universität München, the "AGILO RoboCuppers" (Stulp et al., 2004b), has participated in the mid-size league since 1997. In this league, robots play on a field of approximately 6x8 meters, four against four. The main characteristics of this league is that the robots sense and act locally and autonomously. One of the AGILO robots is depicted in Figure 1.4(a). Experiments have also been conducted in the AGILO simulator, depicted in Figure 1.4(b). These robots will be referred to as 'Pioneer I' and 'Pioneer I (S)' respectively, as these platforms are customized Pioneer I robots from ActivMedia (ActivMedia Robotics, 1998). The hardware and tools of the AGILO RoboCuppers are presented more elaborately in Appendix B.



(a) AGILO RoboCuppers robot (b) AGILO simulator (c) Ulm Sparrow robot

Figure 1.4. The mid-size league soccer domain.

In this adversary domain, performance and efficiency are essential to achieving the goals of the team. Tailoring actions to perform well within the given task context is therefore a necessity. Since it is a multi-robot domain, it also allows us to investigate how actions can be tailored to scenarios with multiple robots. Multi-robot experiments have been conducted in a mixed team with the soccer robots from the Ulm Sparrows (Kraetzschmar et al., 2004), one of which is depicted in Figure 1.4(c).

1.2.2 Service robotics

One of the long-term goals in robotics is to develop robots that can autonomously perform house-hold tasks. Therefore, action models were acquired and applied to a simulated articulated B21 robot in a kitchen. The arms with six degrees of freedom provide this robot with more expressive actions than in the robotic soccer domain. Furthermore, house-hold tasks are less reactive, and require more complex and longer-term planning.

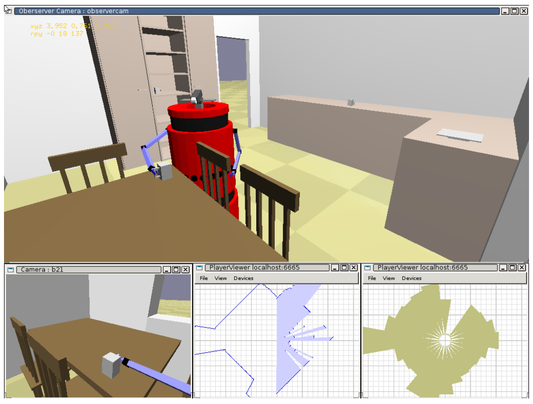
The experiments in this domain were carried out in a simulated kitchen environment (Müller and Beetz, 2006). The simulator is based on the Gazebo simulator of the Player/Stage project (Gerkey et al., 2003). This open-source project develops tools for robot and sensor applications. Gazebo simulates robots, sensors and objects in a three-dimensional environment. The Open Dynamic Engine provides the physical simulation and realistic sensor feedback (Smith, 2004). Player is a network interface and hardware abstraction layer, which the robot's controller uses to communicate with the Gazebo environment. Player facilitates the porting of controllers written in simulation to real robots.

The environment, depicted in Figure 1.5(a) contains a typical kitchen scenario, with furniture and appliances. The positions of the pieces of furniture are static and known. In addition, the environment contains flatware (such as knives, forks, and spoons), cook-ware (pots and pans), and dinnerware (including plates, cups, and bowls). These objects can be recognized and are movable, so the robot can manipulate them. The positions of these objects is known, if they are within the field of view of the robot.

The rich environment and two arms of the robot make the actions much more expressive, which allows for action sequence optimization, to be described in Chapter 5. This domain also allows for more long-term planning than in the extremely dynamic soccer domain.

1.2.3 Arm control

The third domain uses a PowerCube arm from Amtec Robotics (Amtec Robotics, 2005), shown in Figure 1.5(b). Each joint has a brushless servo motor with a Harmonic gear head, and an incremental optical encoder to measure the position. The communication with the computer is done using a high-speed CAN interface. We have mainly included this robot to demonstrate the wide range of domains in which action models can be learned and applied.


(a) The kitchen simulation in Gazebo.

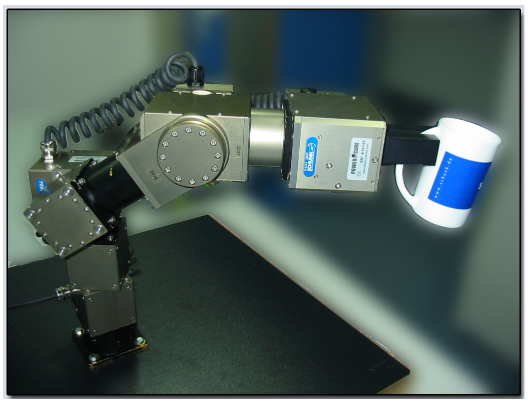

(b) The PowerCube arm.

Figure 1.5. The simulated kitchen environment and the PowerCube arm.

1.3 Contributions

Principle I and Principle II on declarative and procedural knowledge are well established in cognitive science and robotics, as was motivated in Section 1.1. These are the assumptions fundamental to this dissertation. The questions that arise from the ambiguous mapping of declarative to procedural knowledge (Principle III), are essentially the problem statement: How can these questions be answered in a robust and efficient way, without requiring manual programming? The solution to this problem is predictive knowledge (Principle IV), which can be acquired by learning from experience (Principle V). This solution contains the following conceptual contributions:

- Arguing that existing actions can and should be tailored to novel task contexts, rather than designing new customized actions.
- Demonstrating how robots can tailor actions *themselves*, by using predictive knowledge.
- Demonstrating how robots can learn predictive knowledge from observed experience.
- Introducing a novel computational model for the acquisition and application of action models.

Implementing these concepts in the context of several robotic domains and task contexts has led to the following technical contributions:

Action Model Learning Demonstrating how action models can be learned for a variety of robots and tasks. Especially, we investigate how the most can be made of sparse data by including intermediate data between the start and end of each episode. Accurate action models have been learned for all the robots presented in Section 1.2, using tree-based induction.

Subgoal Refinement Free action parameters arise when mapping declarative knowledge to actions. Current controllers often disregard these parameters, which lead to sub-optimal performance. Subgoal refinement explicitly contemplates and optimizes these parameters with respect to the expected performance, predicted by action models.

Subgoal refinement is implemented on three robotic platforms, being the Pioneer I robots of our mid-size league soccer team, the simulated articulated B21 robot, and the PowerCube arm. A variety of action sequences has been optimized, such as sequences of navigation and dribbling actions in the soccer domain, navigation and manipulation actions in the service robotics domain, and reaching movements in the arm control domain. An extensive empirical evaluation demonstrates that subgoal refinement leads to significantly shorter execution times, with smooth motion as a side-effect.

Condition Refinement For humans, specifying symbolic pre-conditions is straightforward, but specifying when they hold in robotic domains is not, due to the complex interaction of the robot's dynamics, the environment, and the action parameterizations. Grounding pre-conditions in real experience is done with condition refinement, which learns to predict action failure, when actions are executed in a novel context with a different goal. Condition refinement is implemented on the Pioneer I (S) robots, using decision trees.

Subgoal Assertion When learned pre-conditions predict that actions will fail, plans are transformed into plans that are predicted to succeed, by introducing a subgoal. The parameterization of this subgoal is constrained by the learned pre-condition, and optimized using subgoal refinement. This procedure is called subgoal assertion.

In the soccer domain, we demonstrate how applying condition refinement and subgoal assertion enables robots to reuse action for novel task contexts. An empirical evaluation verifies that the adapted action is highly successful at achieving novel goals.

Implicit Coordination By reasoning about the states and action models of other robots locally, robots can coordinate their actions to achieve better global behavior. Due to its independence of utility communication, this approach is more robust against communication problems.

Implicit coordination has been implemented in a team of three AGILO robots, as well as in a heterogeneous with both an AGILO and Ulm Sparrow robot. An empirical evaluation shows that implicit coordination is more robust against communication and state estimation failures. Implicit coordination in the heterogeneous team demonstrates that robots with very different hardware and controllers can coordinate with little change to the individual robot controllers.

Because subgoal refinement, condition refinement and subgoal assertion enable robots to autonomously adapt and refine existing actions to novel task contexts, they are a contribution to the field of life-long learning. These methods also bridge the gap between symbolic planning and robot plan execution, and are contributions to both fields. Implicit coordination enables robots to make only local decisions that have effect on the global behavior of several robots, and as such is a contribution to the field of multi-agent systems.

Together, these conceptual and technical contributions provide a framework in which knowledge specified in the controller by humans is complemented, refined and improved with knowledge learned by robots themselves.

1.4 Outline

The following is a synopsis of the individual chapters of this dissertation.

Chapter 2 - Computational Model. This chapter introduces the terminology, concepts and methodology used throughout this dissertation. It also presents an overview of the system.

Chapter 3 - Related Work. Work related to action selection schemes, forward models and action models will be discussed. Both cognitive science and robotics research are treated. Work related to specific applications of action models will be discussed in the respective chapters.

Chapter 4 - Learning Action Models. Action models are acquired by learning them from observed experience. In this chapter, we describe how the necessary experience is gathered, and how generalized models can be learned from this data.

Chapter 5 - Task Context: Action Sequences. The first application of action models is to tailor actions to perform well within a given action sequence. The method with which this is done is called *subgoal refinement*.

Chapter 6 - Task Context: Task Variants. In this chapter we present *subgoal assertion* and *action refinement* in which action models used to parameterize available actions so that they can be reused for a new task variant.

Chapter 7 - Task Context: Multiple Robots. Action prediction models can be used to coordinate the actions of multiple robots. By predicting the performance of other robots, a robot can adapt its actions accordingly. This is called *implicit coordination*.

Chapter 8 - Conclusion. The content of this dissertation is summarized in this conclusion. Note that directions for future research and related work has been discussed in Chapters 4 to 7.

Chapters 4 to 7 describe how action models are acquired and applied on the robots, and contain the technical contributions. These four chapters have the same structure. After an introductory section, the computational model is presented. The following sections in these chapters then explain how the computational model was implemented. After presenting the empirical evaluation conducted on the robots, work specifically related to this chapter is discussed and compared with our work. The conclusion contains a summary of the chapter and directions for future work.

2. Computational Model

“Before turning to those mental aspects of the matter which present the greatest difficulties, let the inquirer begin by mastering more elementary problems.”

Sherlock Holmes – Arthur Conan Doyle

In this chapter, we will introduce and formalize the basic concepts and terminology used throughout this dissertation. The relevant concepts are separated into *entities*, which are data structures, and *processes*, which manipulate the entities. Examples from the robotic soccer domain will be used throughout. The next section introduces the *dynamic system model*, which desribes the interaction of an agent with its environement, and the role of the controller within the agent. In Section 2.2, we demonstrate that the concepts of durative actions and action selection can elegantly be described using the dynamic system model. At the end this of this chapter we will give an overview of the system presented in this dissertation.

2.1 Dynamic System Model

The standard model for control theory is the dynamic system model by Dean and Wellmann (1991). In this model the world changes through the interaction of two processes: the Controlled Process and the Controlling Process, as depicted in Figure 2.1. In robotics, the controlled process is the behavior and perception of the robot in its environment. Given this model, the two main steps in designing the controlling process on the robot are specifying the state estimation and controller.

2.1.1 Controlled process

In the controlled process, the Environment Process is simply either the physical world the robot is embodied in, be it real or simulated. The evolution of the environment process is

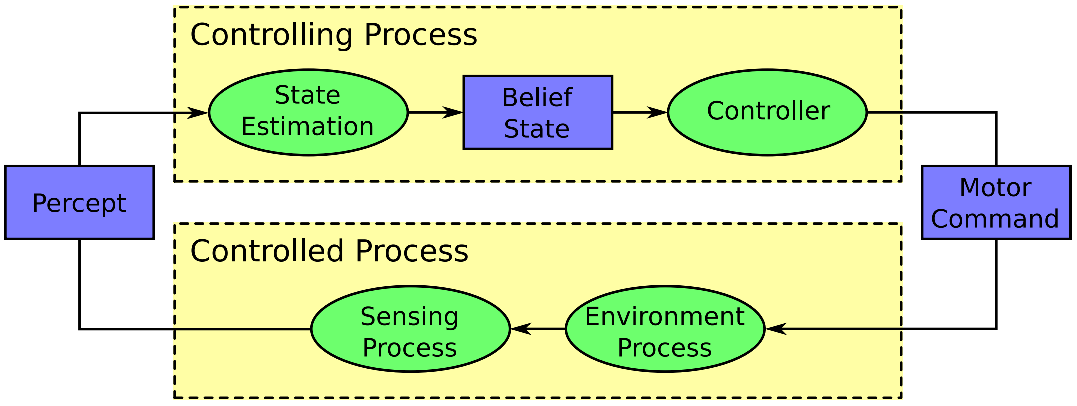


Figure 2.1. Dynamic system model

represented by a set of *state variables* that have changing values. The state of the environment can be influenced by applying Motor Commands to it[1]. Motor commands directly set some of the state variables in the environment process and indirectly other ones. The affected state variables are called the *controllable* state variables. For instance, the robot can set the translational and rotational velocity directly, causing the robot to move, thereby indirectly influencing future poses of the robot.

The robots used in this dissertation send motor commands to a hardware component at regular intervals. For instance, the motor command for the soccer playing robots with differential drive is $[v, \dot{\phi}, kick]$, which specify the translational and rotational speed. This motor command is processed by a hardware component and converted to voltage levels for both motors. In the dynamic system model, the hardware component and its processing is part of the controlled process, not the controlling process. The only interface the controller has to influencing the world's state is the motor command.

The Sensing Process represents the sensor of the robot, which are embedded in the environment process. The unprocessed data structures these sensors generate are called Percepts. For the robot, often only a subset of the state variables is *observable* to its perceptive system, and only these variables are encoded in the percept. The percepts of our soccer robots for instance, are camera images, odometry, and messages received from other robots. Note that these percepts do not arrive as one single data structure, but arrive and are processed asynchronously.

[1]In the dynamic system model, motor commands are actually called control signals. We prefer the term 'motor command', as it emphasizes that all the control signals in this dissertation are sent to the motor system of the robot.

2.1.2 Controlling process

The controlling process' task is to produce a sequence of control motor commands that affect the environment, for instance to achieve a certain goal. To generate motor commands that direct the environment to a desired future state, the controlled process must often first know the current state of the environment. This state is estimated from the percepts with State Estimation. For instance, the soccer robots use a cooperative probabilistic state estimation with opponent tracking, which uses the available percepts, being camera images, odometry, and communication with teammate robots (Beetz et al., 2004).

The output of the state estimation is a Belief State. The belief state represents the robot's beliefs about the current values of the state variables in the environment (Schmitt et al., 2002; Utz et al., 2004). Due to limitations of sensors and state estimation, the true state of the world cannot be determined with full certainty and accuracy. The controlling system does not know the state of the world, but rather has beliefs about it, hence the term 'belief' state. The term *world state* should rather be used for the actual state of the world, and the *world model* is the description of all possible belief states. The belief state of the soccer robots contains observable state variables related to their own pose on the field, as well as those of its teammates and opponents. The position of the ball is stored, as well as any unidentified obstacles on the field.

The Controller takes a belief state as an input, and returns a motor command. This dissertation focusses on the designing and learning effective controllers. If the controller is not purely reactive, it also has a internal state, which is described in terms of *internal* state variables. Examples are the current goal, or the sequence of actions it is committed to executing, as well as their parameterizations. Furthermore, there is a distinction between *direct* and *derived* state variables. Direct state variables are directly provided by state estimation (e.g. position of ball and myself), whereas derived state variables are computed by composing direct variables (e.g. distance to ball). No extra information is contained in derived variables, but if chosen well, derived variables are better correlated to the control task. This will be explained more elaborately in Section 4.1.1.

Summarizing, percepts are acquired through sensors embedded in the environment. State estimation estimates the observable state variables from the percepts, and stores them in the belief state. The controller takes the belief states, and determines a motor command that will direct the environment into a desired goal state. These motor commands are sent to the controlled process. For example, a soccer robot uses its camera (sensing process) to capture images (percepts), converts them into ball and robot positions on the field (belief state), and gives velocity commands (motor commands) to the motors, for instance to dribble the ball (goal).

2.2 Durative Actions and Action Selection

One way to design controllers is through direct programming. The designer contemplates the domain and the task to be executed, and fully specifies which action should be executed in which state. For the game of tic-tac-toe, it is feasible, though tedious, to specify for each of the 765 legal states, which move to play next. A more realistic example is designing a PID controller to control the temperature in a room. The percept (current temperature) and 'motor command' (power for the heater) are continuous, so enumerating all states and commands would be impossible. Nevertheless, a relatively simple function suffices to map each input to an output.

When controllers perform tasks in complex dynamic domains this monolithic approach can become very tedious and error-prone. Imagine enumerating all possible situations in robotic soccer, and specifying the desired velocity command for each of them. Designing a single PID controller that can play soccer is just as infeasible.

The predominant approach in robotics to solve this problem is to first design or learn a set of actions (Principle II), and then design or learn an action selection module, that chooses the appropriate action given the current context (Principle I). A schematic overview of the organization of actions and action selection is depicted in Figure 2.2.

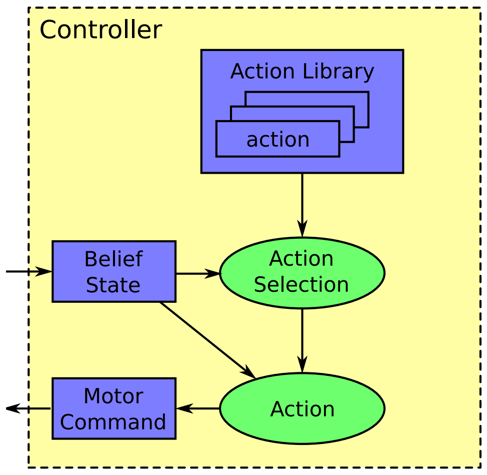


Figure 2.2. System overview of actions and action selection in the dynamic system model controller

An Action is a control programs that produce streams of motor commands, based on the parameters with which it is called. The parameters are a subset of the direct observable variables in the belief states and internal variables, or variables derived from them. Note that in Figure 2.2, actions are depicted both as entities (boxes) and processes (ovals). On the one hand, actions are processes, as they transform belief states into motor commands. On the other hand, the action selection considers actions to be resources or entities it can manipulate and reason about.

Actions can be executed in the real continuous world, because the motor commands they generate can directly be dispatched to a hardware component. The parameters of an action are either observed variables, describing part of the current belief state, or internal variables, describing the current subgoal. In this dissertation, actions themselves have no internal state; they are purely reactive. Any persistent information must be stored outside of the action. As an example, the signature of the `goToPose` action is `goToPose(`$x, y, \phi, v, \quad x_g, y_g, \phi_g, v_g$`)` It navigates the robot from the current dynamic pose $[x, y, \phi, v]$, stored in the belief state, to a future goal pose $[x_g, y_g, \phi_g, v_g]$, stored in the internal state. It does so by returning motor commands $[v, \dot{\phi}]$, representing the translational and rotational velocity of the robot.

The main resource of an action based controller is the Action Library. Action libraries contain a set of actions that are frequently used within a given domain. If actions are specified general, and apply to a large set of the state space, only a few actions are needed to execute all possible tasks in a certain domain.

Table 2.1 lists the actions used in this dissertation. The actions have been implemented in C++ or Python (PowerCube). The action parameters in the signatures have been partitioned, based on whether they hold in the current state of the world or if they specify the target the robot wants to achieve. Note that the first are observable variables, and the second are internal variables. Although learning and applying action models is independent of actual action implementations, we list their implementations for completeness in Appendix A

Robot	Action	Action Parameters		Motor
		Observed	Internal	Comm.
AGILO	`goToPose`	x, y, ϕ, v	x_g, y_g, ϕ_g, v_g	$v, \dot{\phi}$
Ulm Sparrow	`goToPosition`	x, y, ϕ, v	x_g, y_g, v_g	$v, \dot{\phi}$
B21	`goToPose`	x, y, ϕ, v	x_g, y_g, ϕ_g, v_g	$v, \dot{\phi}$
	`reach`	x, y, z, ax, ay, az	$x_g, y_g, z_g, ax_g, ay_g, az_g$	?
PowerCube	`reach`	$\theta^a, \dot{\theta}^a, \theta^b, \dot{\theta}^b$	$\theta^a_g, \dot{\theta}^a_g, \theta^b_g, \dot{\theta}^b_g$	I^1, I^2

Table 2.1. List of actions used in the application domains

This list 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 exactly the goal of this dissertation to show how only a few actions can be reused and customized to perform well in varying task contexts. In Chapters 5 to 7, we will demonstrate how the robots parameterize this action to approach the ball, dribble it, navigate efficiently through way-points, and regain ball possession in a team of robots.

The Action Selection module selects the appropriate action in a given context. In Section 3.1, various approaches to designing and learning action selection modules will be presented.

2.2.1 Advantages of durative actions

When introducing Principle II in Section 1.1, some of the advantages of durative actions were discussed. We will repeat them here more elaborately, using the conceptualization introduced in this chapter.

Actions themselves are controllers, as their input is also a belief states, or a subset of variables from the belief state, and return motor commands[2]. However, since actions only apply to certain limited task contexts, they are easier to design or learn than a controller that must be able to deal with all possible contexts (Haruno et al., 1999; Jacobs and Jordan, 1993). For instance, a soccer robot might have the action `dribbleBall`, that only applies in states where the robot is in possession of the ball. Designing or learning one monolithic controller that can play soccer might be infeasible, but designing or learning an action that can dribble is not.

Another advantage of durative actions is that they provide an intermediate temporal abstraction between high-level goals and low-level motor commands. Instead of having to directly select motor commands every few milliseconds, the action selection module selects actions every few seconds. Furthermore, actions provide a conceptual abstraction. Because actions are designed with a certain task and goal in mind, they can be selected based on *what* they do, thereby abstracting away from *how* they do it. For instance, the name of the action `dribbleBall` alone already gives a clear indication of what it is intended to do, although it is unknown, and for action selection purposes irrelevant, how it actually achieves what its name indicates. These two abstractions enable the action selection module to be specified on a high level of abstraction.

Action based systems are also more adaptive. Single or several actions can be adapted to new environments without having to change the action selection module. Of course, this only holds if the abstract functionality of the actions remains the same, and the implementation

[2]The terms controller and action can in principle be used interchangeably. In this dissertation, only the top-level controller in the dynamic system model is referred to as the controller, and the controllers at lower levels are always referred to as actions.

of the action is hidden from the action selection module. These benefits are well-known in Software Engineering, where this design approach is known as the Bridge Pattern (Bruegge and Dutoit, 2003).

Let us now summarize the benefits of using actions and action selection in controller design:

1. Actions apply to only a subset of tasks, which facilitates learning and designing them.
2. Actions provide a temporal abstraction between high-level goals and low-level motor commands.
3. Actions provide a conceptual abstraction.
4. This temporal and conceptual abstraction enables action selection at a more abstract level, which facilitates controller design.
5. Actions can be adapted, without affecting the action selection module.

For animals with complex motor capabilities, especially the first reason has lead to the use of *inverse models*, which is nature's equivalent of an action (Haruno et al., 1999).

2.3 Guide to the Remainder of the Dissertation

In Section 1.1, some of the questions that remain when mapping declarative to procedural knowledge (Principle III) were discussed. For efficient control in multi-robot environments, any controller will have to answer these questions. Figure 2.3 depicts an overview of the system with an action-based controller from the dynamic system model, along with the questions it must answer. These questions only arise in certain task contexts, along with which they are listed.

The key to answering these questions is using predictive knowledge (Principle IV), which is compiled into Action Models. Action models allow agents to reason about what their actions can do, and how well. They are called with the same parameters as the corresponding action, and therefore have the same signature. Instead of returning a motor command, action models return the expected performance of executing this action, given these parameters. In this dissertation, the most frequently used performance measure is execution duration. The action models used in the different application domains will be listed in Section 4.1.3. Some examples of action models will be presented in Sections 4.2 and 6.2.

The first step in the system is to acquire action models for each action in the action library. Action models are learned from observed experience (Principle V). Gathering training examples can be done in idle time, when the agent is not required to perform other tasks. Learning these models compiles a wealth of experience into a concise model, which generalizes over

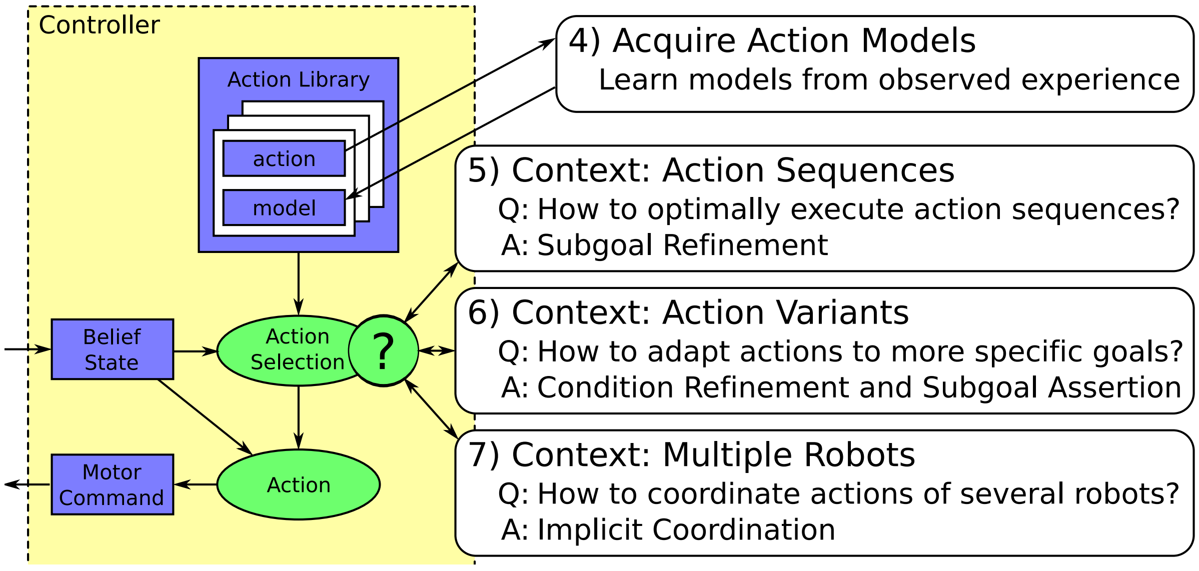

Controller
Action Library
action
model
Belief State
Action Selection
?
Motor Command
Action
4) Acquire Action Models
Learn models from observed experience
5) Context: Action Sequences
Q: How to optimally execute action sequences?
A: Subgoal Refinement
6) Context: Action Variants
Q: How to adapt actions to more specific goals?
A: Condition Refinement and Subgoal Assertion
7) Context: Multiple Robots
Q: How to coordinate actions of several robots?
A: Implicit Coordination

Figure 2.3. System overview for the acquisition and application of action models. Numbers correspond to Chapter numbering.

situations not yet experienced. Action models are also stored in the action library, alongside their corresponding action. At operation time, these models predict the expected outcome and performance of actions, at negligible computational cost.

Then, these action models are used to answer the questions that remain when mapping declarative knowledge to procedural knowledge in different task contexts, as can be seen in Figure 2.3.

3. Related Work

This chapter will present related work on action selection schemes, as well as the the acquisition of action models in nature and robotics. The difference between forward models, action models and reinforcement learning values will be explained. Comparisons between this related work and the system presented in this dissertation will be made in the individual chapters on the acquisition (Chapter 4) and applications (Chapters 5 through 7) of action models, after the relevant system modules have been presented.

3.1 Action Selection Schemes

In Section 2.2 the general computational model for controllers with durative actions and an action selection module were presented. We will now briefly present four well-known approaches to designing action selection modules. They are introduced here for future reference; a more elaborate explanation of their advantages, disadvantages and relation to this research will be given throughout the dissertation, for instance in Sections 4.4.1, 5.2.3, and 5.5.1.

3.1.1 Direct programming

Actions provide a level of abstraction that we can reason about, similarly to the conscious deliberation of our own actions. This makes direct programming of the action selection module feasible. The most straight-forward method is to code the action selection directly into the programming language used for the robot's controller. Alternatively, a *behavior language* that is tailored to developing controllers can be used. For instance, several languages exist that allow controllers to be designed in terms of state charts. Examples of this approach are (Lötzsch et al., 2004), in which state charts are coded in XML, or (Murray, 2001; Arai and Stolzenburg, 2002; Obst, 2002) in which the same is done with UML. The benefit of this approach is that since the designer has hand-coded everything, the displayed behavior can be explained in terms of the designer's knowledge and intentions. This can facilitate behavior debugging.

Of course, the disadvantage is that the designer has to hand-code the action selection module, which is tedious. It is also error-prone, as the designer cannot be expected to foresee each possible situation and specify an appropriate response, although these situations might occur in the real world. Also, this approach does not scale well. The more complex the environment, the more actions and interactions between actions must be taken into account when designing action selection.

3.1.2 Motion blending

In motion blending approaches, there is no exclusive action selection, as all actions constantly compute a motor command. The final motor command the controller returns is computed by interpolating between the various motor commands, with a certain weighting scheme. The benefit of this approach is that there are no discrete transitions between movements, which is important if fluency of motion is required. Examples of controllers that use motion blending are (Jaeger and Christaller, 1998; Utz et al., 2005; Saffiotti et al., 1993). Most behavior-based approaches use motion blending as well (Arkin, 1998; Brooks, 1986).

3.1.3 Hierarchical Reinforcement Learning

Another possibility is to learn the action selection module with Reinforcement Learning (RL). In this approach, controllers are learned by optimizing a reward function. At each discrete time step, the controller returns the motor command that maximizes the future discounted reward, also called *value*. The problem of learning to choose commands is thus converted to learning to predict the value of each state and action. Temporal difference learning has proven to be an efficient method to learn these values. However, monolithic RL, which learns one value mapping for the entire domain, does not scale to complex tasks (Barto and Mahadevan, 2003).

Hierarchical Reinforcement Learning solves part of the problem by defining a set of actions that can achieve subsets of the entire task. The value for each primitive action in these actions is then learned, as well as the value for executing an entire action in a certain state. Because these high-level and low-level values depend on each other, even better results can be achieved when both values are learned simultaneously (Dietterich, 2000; Kleiner et al., 2002). Section 4.4.1 will discuss the advantages and disadvantages of Hierarchical Reinforcement Learning.

3.1.4 Planning

In plan-based control, the robot explicitly reasons about the applicability and effects of actions to select a sequence of actions to achieve a goal. An important aspect of plan-based robot control is that robots contemplate and commit to a sequence of action *prior* to execution. This allows the controller to consider interactions between future actions, and resolve conflicting goals in advance, before they are encountered on-line. In recent years, a number of autonomous robots, including Minerva (Beetz, 2001), WITAS (Doherty et al., 2000), and Chip (Firby et al., 1996), have shown impressive performance in long term demonstrations. The use of planning enables these robots to flexibly interleave complex and interacting tasks, exploit opportunities, and optimize their intended course of action.

To reason about action sequences, the controller must be able to project them into the future internally, without actually executing them in the real world. Planning approaches therefore define the pre- and post-conditions of each actions. These declarative components specify when the action is applicable, and what its effects are when executed. Planning systems take a set of actions and a goal that has the same format as a pre-condition, and generate a sequence of actions that achieve the goal. In this sequence, the post-condition of each action satisfies the pre-condition of the subsequent action. Furthermore, the pre-condition of the first action is satisfied by the current situation, and the post-condition of the last action must satisfy the goal. This represents a valid plan to achieve the goal. As we shall see in Section 5.2.1 the declarative knowledge contained in the pre- and post-conditions can be easily human-specified.

3.1.5 Different terminologies for actions

Structuring controllers in terms of actions and action selection modules is so effective in reducing action search spaces, that it has co-evolved in many of the approaches to designing controllers, as we saw in Section 2.2. Table 3.1 lists some examples of these approaches, and the terminology for motor command and action they use. Note that Cognitive Science has also been included. Although the main objective of this field is not to design controllers, it certainly does analyze the control systems of humans.

In this dissertation, a durative action is simply referred to as an “action” for reasons of brevity. The term “motor command” refers to smallest unit of action as a reminder that they are very close to the execution on a motorized hardware system.

Domain		
Reference	**Motor Command**	**Action**
Control Theory		
(Dean and Wellmann, 1991)	Control Signal	Controller
(Jacobs and Jordan, 1993)	Control Signal	Controller
(Qin and Badgwell, 1998)	Controllable Variables	Controller
Direct Programming		
(Murray, 2001)	Command	Skill
(Lötzsch et al., 2004)	Action	Option
Behavior based / Motion Blending		
(Brooks, 1986)	Motion Command	Module/Behavior
(Jaeger and Christaller, 1998)	Motor Command	Behavior
Reinforcement Learning		
(Sutton et al., 1999)	Primitive Action	Option
(Andre and Russell, 2001)	Action	HAM, PHAM
(Dietterich, 2000)	Action	Subtask
(Ryan, 2004)	Primitive Action	Behavior
Planning		
(Fikes and Nilsson, 1971)	Low-level action	Routine
(Nilsson, 1994)	Primitive Action	T-R Program
(Ryan, 2004)	Primitive Action	Behavior
(Belker, 2004)	Motor Commands	Action
(Bouguerra and Karlsson, 2005)	Action	Executable Action
(Cambon et al., 2004)	Motion	Action
Forward Models		
(Wolpert and Flanagan, 2001)	Motor Command	Inverse Model
(Dearden and Demiris, 2005)	Motor Command	Inverse Model
(Jordan and Rumelhart, 1992)	Action	Inverse Model

Table 3.1. Differing terminologies for different approaches to designing action-based controllers.

3.2 Action Models

In this section, we will first discuss related work on forward models in nature and robotics. The difference between forward models and action models will be explained, and some uses of action models in robotics presented.

3.2.1 Natural forward models

In cognitive science there is a distinction between inverse models, which map desired consequences to motor commands, and forward models, which map motor commands to their effects. Forward models make predictions, because current motor commands are mapped to future outcomes.

Helmholtz (1896) provided the first proof for the existence of forward models in humans, in the context of object localization. Due to eye movements, the projections of objects in the world on the retina are constantly moving. To acquire a stable image of the world the brain takes the position of the eye in its socket into account. Instead of sensing the eye's position, its position is predicted by using information from the eye muscles. Helmholtz's simple and ingenious experiment demonstrates this. If one eye is closed, and the position of the other eye in the socket is moved artificially by pressing it with your finger, the world seems to be moving, because the compensating prediction based on the motor command sent to the eye's muscles is lacking.

In a more recent experiment, subjects were asked to follow the voluntarily reaching movements of their arm with their eyes (Ariff et al., 2002). If the arm is hidden from the subject's view, the subjects make saccadic movements movements to a location that predicted the position of their hand 196 ms in the future.

Especially in the last decade, many new discoveries about how forward models are learned and used have been made (Wolpert and Flanagan, 2001; Wolpert and Ghahramani, 2000). This section will present an overview of these results.

Forward models are learned

Forward models are not entities that are fixed at birth, but that must rather be learned and updated through experience. This allows forward models to be learned for new action contexts, or for newly acquired actions. Supervised learning can be applied, because prediction errors can easily be acquired by comparing the predicted and actual outcome of a motor command. The neural mechanisms behind such predictive learning are partially understood in electric fish (Bell et al., 1997). It is hypothesized that "body babbling" is a strategy to actively acquire training data to learn such models (Rao et al., 2005).

Humans actually learn the forward model of an action *before* the final inverse model is learned (Flanagan et al., 2003). So, the brain learns to predict the effects of an action before perfecting the execution of the action itself. In the approach presented in this dissertation a similar procedure is described. First action models are learned from observed experience for the actions in an action library. These action models can then be used to tailor actions to task contexts, such as action sequences or multiple robots.

Widespread use of forward models in motor control

Humans use forward models in many task contexts. Some examples will be presented in this section. Optimal control and social interaction, the items marked with a *****, are applications of forward models that have been implemented in this dissertation as well. They will be discussed in more detail in Chapter 5 and Chapter 7 respectively.

State estimation Accurate control of the body requires on knowing the body's state, such as the joint angles, and the positions and velocities of body parts. Due to neural transmission and processing, sensory signals that provide information about the body's state have considerable delay. Especially for fast movements, a more timely estimation of the body's state is essential. Alternatively, predictions based on motor commands can be used to update the state, even before the movement is executed (Wolpert and Flanagan, 2001). In control, the Kalman filter (Kalman, 1960) is an example where state estimation is also performed with both motor and sensor updates.

Sensory cancellation Prediction also allows sensory information to be filtered, for instance to cancel out the sensory effects caused by self motion. For example, it is impossible to tickle oneself, because the expected sensory consequences of this motion, predicted with forward models, are subtracted from the actual sensory feedback. In an recent experiment, subjects tickled themselves through a robot interface (Wolpert and Flanagan, 2001). An arbitrary delay between the tickle command and actual tickling could be introduced through the robot interface. It was shown that the larger the delay, the more 'ticklish' the percept, presumably to a reduction in the ability to cancel the sensory feedback based on the motor command.

Context estimation Different contexts require different behaviors. Humans are very good at selecting the appropriate behavior, even under uncertain conditions. One explanation is that several inverse models are tested for their appropriateness in parallel. For example, when initially lifting an object of unknown weight (is the box empty or full?), the forward models of the inverse models for lifting both light and heavy objects are active. Once lifting commences, the error between the prediction and the actual movement is measured for each forward model. The inverse model corresponding to the forward model that generates the lowest error is then chosen as the appropriate controller. Several of these paired forward-inverse models are integrated in the MOdular Selection and Identification for Control (MOSAIC) framework (Haruno et al., 2001).

Optimal control * Although there are infinitely many ways to perform most tasks, tasks are usually solved with highly stereotyped movement patterns (Wolpert and Ghahramani, 2000). The optimal control framework assumes that these typical patterns are those that minimize a certain cost function. In cognitive science, the challenge is to reverse-engineer this cost function, given the motion patterns found in empirical studies. For instance, for reaching movements there exist optimal control models that optimize the smoothness of the trajectory (Flash and Hogan, 1985), smoothness of the torque commands (Uno et al., 1989) and variability of movement (Harris and Wolpert, 1998).

Social interaction * Wolpert et al. (2003) hypothesize that forward models form the basis of social interaction and imitation. There are many similarities between the motor loop and the social interaction loop. In the motor loop, a motor command changes my body's state, whereas a communicative command (e.g. speech, gesture) changes the mental state of another. Possibly, forward models are used in predicting the change in mental state due to my commands too. It may be that the same computational mechanisms which developed for sensorimotor prediction have adapted for other cognitive functions.

Imitation Once the responsible forward models for executing an action have been recognized, imitating the action is relatively straightforward: activate the inverse models belonging to these forward models in the same order as the forward models were recognized. Wolpert et al. (2003) describe a hierarchical version of the MOSAIC system that models this process.

3.2.2 Robotic forward models

The widespread use of forward models in human motor control has drawn the attention of control and robotics community. Jordan and Rumelhart (1992) introduced Distal Learning, the first method to explicitly use forward models in a controller. The distal supervised learning problem is defined by *intentions*, that specify what the controller wants to achieve, *motor commands*, with which the controller can influence the environment, and *outcomes*, the result of executing motor commands in the real world. The problem is that the inverse model has to map intentions to motor commands, but has no target values for these motor commands. There are target values for the outcomes, but these cannot be influenced directly by the inverse model, which is why they are called *distal*. Because the target values are distal, learning the inverse model cannot be done with supervised learning. The key to solving this problem is learning an internal forward model, which maps motor commands to outcomes. Forward models *can* be learned using supervised learning, because they are a mapping from actions to proximal target outcomes. This section has demonstrated how data can be gathered to learn this mapping.

The resulting composite learning system with inverse and forward models can be treated as a supervised learning problem, which can be learned with any supervised learning algorithm.

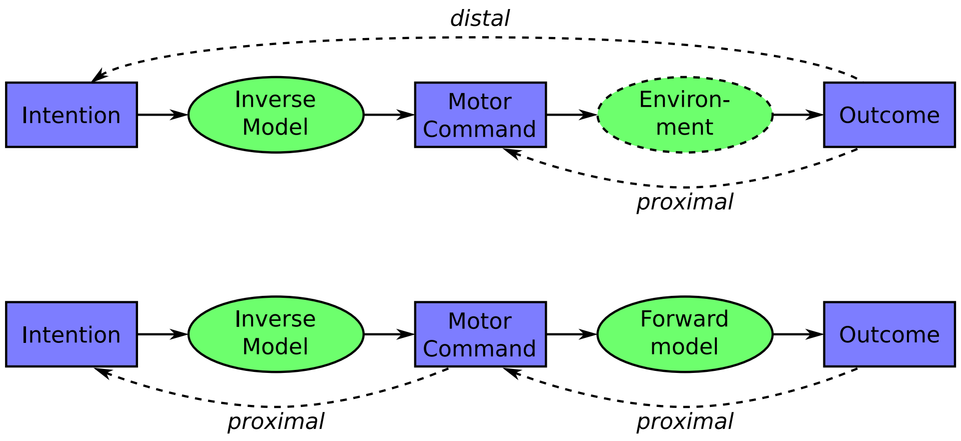


Figure 3.1. The distal learning problem, with distal target values (above). Forward models are the key to solving the problem (below).

Recently, robotic forward models have also been learned using Bayesian networks, as described by Dearden and Demiris (2005). The benefit of Bayesian networks is that they allow the causal nature of a robot's control system to be modelled using a probabilistic framework. Infantes et al. (2006) describes recent work at another group that also includes the use of dynamic Bayesian networks.

In the networks used, nodes represent motor commands, robot states or observations, and edges represent causal associations between these nodes. Motor commands cause changes in the robot's state, which is hidden, and this in turn causes changes in the observations, which are accessible through the vision system. The structure and parameters of the network can be learned by data acquired through motor babbling, similarly to the approach described in Section 4.1.

A nice side effect of Bayesian networks is that the delay with which a motor command actually changes the robot's state and observations is not fixed. By determining the log-likelihood for varying delays, Dearden and Demiris could determine that issuing a velocity command lead to an observed velocity 550ms later. Such delays must be taken into account when other mappings from motor command to observation are learned, for instance when learning an inverse model from human examples, as in (Buck, 2003), where the dead time is 300ms.

3.2.3 Robotic action models

Forward models predict the outcome of executing a motor command, whereas action models predict the cost of continually executing a durative action. Forward models make prediction on a time-scale of several 100ms, whereas action models predict the performance or outcome of an action on completion, possibly several seconds or more in the future. Just as it is hypothesized that forward models form the basis of social interaction and imitation (Wolpert et al., 2003), we hypothesize that forward models could be reused to yield action models in the brain.

In principle, forward models can be called recursively to emulate an action model. Instead of making a prediction only one time step ahead, a sequence of motor commands could be used to update a simulated state n time steps in the future. If this sequence of motor commands is generated by an inverse model, given the simulated state, we are effectively simulating the temporally extended effects of the inverse model on the current state. This could be used to determine how long the inverse model must be executed to achieve a certain state, or if this state can be achieved at all. A drawback of this approach is that the uncertainty of the prediction grows with each recursive call. Dearden (2006) has demonstrated empirically that this accumulated uncertainty prevents this approach from being used in practice. Furthermore, the abstract effects of

Balac (2002) proposes the ERA (Exploration and Regression tree induction to produce Action models) system, which learns action models from observed data by training regression trees. In this work, a robot learns the velocity with which it can travel over terrains with different roughness properties. This knowledge is used to improve navigation plans. The interesting aspect of this work is that for one action, different models are learned for different contexts.

Haigh (1998) uses regression trees as well to learn cost models for indoor navigation actions. These models take features such as the time of day into account as well. This is useful to predict the crowdedness of hallways, and thus the duration of navigation. These models are used to compute the best route in the office environment. In another application, search control rules for the planning rules are derived from the regression tree rules.

Belker (2004) describes how action models are learned for navigation actions using model trees and neural networks, and also stresses the importance of defining an appropriate feature space. Since the emphasis in this work is on indoor navigation and obstacle avoidance, features regarding the number of passages and their width (narrow vs. wide) are also included in the feature space. The use of these models will be discussed more elaborately in Section 5.5.3. Buck et al. (2002b) has a similar approach with neural networks.

3.2.4 Reinforcement Learning

In Supervised Learning, a teacher provides the target value vector for each input value vector, for instance the appropriate action given a set of observations. For many problems in control, the target action is not available, as this is exactly what we want to learn. As we saw, distal learning is one solution to this problem. Reinforcement Learning (RL) solves the problem by requiring the teacher to provide far less information: instead of providing a target action at each time step, the teacher must only provide rewards at desirable state. The problem in Reinforcement Learning is to find the sequence of actions that optimizes the accumulated reward over time.

Most RL algorithm simplify this problem by not looking for the best sequence of actions (policy) directly, but attributing a value V to each state, or a value Q to each state-action pair (Sutton and Barto, 1998). Here, we will focus on Q-learning. At each time step, exploration strategies aside, the controller simply chooses the action with the highest Q-value. Although this simplifies action selection greatly, the problem of attributing values state-action pairs such that the accumulated reward over time is optimized remains. The main idea behind RL algorithms is that these values can be learned, that is updated incrementally by backtracking the chosen action sequence each time a reward is found. Many improvements on this initial idea have been made, such as intelligent updating, intelligent exploration, updating values off-line, allowing continuous state and action spaces, etc. We will not elaborate on them here.

Even with these improvement, RL algorithms only solve problems with small discrete action and state spaces, because values must be saved for each state-action tuple. This number increases exponentially with the number of dimensions in the state and action space. Recent attempts to combat this curse of dimensionality in RL have turned to principled ways of exploiting temporal abstraction (Barto and Mahadevan, 2003). Several of these *Hierarchical Reinforcement Learning* methods, e.g. (Programmable) Hierarchical Abstract Machines (Parr, 1998; Andre and Russell, 2001), MAXQ (Dietterich, 2000), and Options (Sutton et al., 1999). All these approaches use the concept of actions (called ‘machines’, ‘subtasks’, or ‘options’ respectively).

The important aspect of Reinforcement Learning for this dissertation is that V- and Q-values can be thought of as compiled action models. In RL, the policy is the action, and the value is the predicted reward at some future time. A comparison between values and action models will be made in Section 4.4.1.

3.2.5 Terminology

For completeness, Table 3.2 lists the different terminologies for action effects (*what*) and performance prediction (*how well*)in different approaches. It is a repetition of Table 3.1, where approaches that have no concept of action prediction have been excluded.

Domain			
		Prediction	
Reference	**Action**	**Effects**	**Action Model**
Control Theory			
(Qin and Badgwell, 1998)	Controller	Process Model	—
Reinforcement Learning			
(Sutton et al., 1999)	Option	—	Q-Value
(Andre and Russell, 2001)	HAM, PHAM	—	Q-Value
(Dietterich, 2000)	Subtask	—	Q-Value
(Ryan, 2004)	Behavior	Effects	Q-Value
Planning			
(Fikes and Nilsson, 1971)	Routine	Effects	—
(Nilsson, 1994)	T-R Program	Effects	—
(Ryan, 2004)	Behavior	Effects	Q-Value
(Belker, 2004)	Action	Effects	Action Model
(Bouguerra and Karlsson, 2005)	Executable Action	Effects	—
(Cambon et al., 2004)	Action	Effects	—
Forward Models			
(Wolpert and Flanagan, 2001)	Inverse Model	Forward Model	—
(Dearden and Demiris, 2005)	Inverse Model	Forward Model	—
(Jordan and Rumelhart, 1992)	Inverse Model	Forward Model	—

Table 3.2. Differing terminologies for different approaches to designing skill-based controllers.

3.3 Cognitive Systems

Action models enable robots to reason about the outcome and performance of their actions. Such reflective capabilities is essential for any cognitive system. In this section, we will present work related to the overall approach of designing and implementing cognitive systems.

The overview paper “Systems That Know What They’re Doing” (Brachman, 2002), explains the DARPA Information Processing Technology Office’s goal to transform systems which simply react to inputs to systems which are cognitive. In the proposed architecture, a differentiation between reactive, deliberative, reflective and a self-awareness processes is made. Reactive processes are simple reflexes and automated behavior routines whose execution does not need conscious effort. The bulk of decision making is performed by deliberative processing, whereas reflective processes contemplate this decision making process to reflect on alternative approaches. Self-awareness, the ability to realize that we are individuals with different experiences, capabilities and goals, is an additional capability that enables even more powerful reflection. These processes will allow systems to perform more robustly and independently in application domains such as information extraction, networking and communications, or computational envisioning. Action models enable systems to deliberate the outcome of there actions, and reflect on alternative actions or action parameterizations on-line.

Cognitive Systems for Cognitive Assistants (CoSy) (Cosy, 2004) is a project whose goal it is to study cognitive submodules in the context of an integrated system. The methodology in this project is to iteratively determine and implement intermediate steps, without losing track of the ultimate goal of human-like performance. Another key principle is to understand which approach is best in which context: nature or nurture, reactive or deliberative, explicit or implicit representation. The projects also stresses the importance of finding representations that allow powerful interactions between submodules. In this sense, action models can be thought of as very powerful representations, as they facilitate control, state estimation and many other aspects of cognitive systems.

The Modular Selection And Identification for Control (MOSAIC) architecture (Haruno et al., 2001) 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. The inverse models are learned during the task, and the forward models are used to select the appropriate inverse model in a certain context. However, this architecture has not been designed for robot control. We are not aware of (robotic) controllers in which prediction 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.

4. Learning Action Models

"Skilled motor behavior relies on the brain learning both to control the body and predict the consequences of this control"

(Flanagan et al., 2003)

As this quote implies, the key to answering the questions related to effectively and efficiently executing actions in different, possibly novel, task contexts is prediction. As we saw in Section 3.2.1, humans do exactly this, by learning forward models, and extensively using them in various motor control tasks. For instance, forward models are used to improve state estimation, estimate contexts, optimize control (Helmholtz, 1896; Wolpert and Ghahramani, 2000; Wolpert and Flanagan, 2001; Ariff et al., 2002), and are possibly the basis of social interaction and imitation (Haruno et al., 2001). In some holistic architectures of cognition and motor control, predictive knowledge plays a more important role than declarative knowledge (Hawkins and Blakeslee, 2004; Grossberg, 1987; Haruno et al., 2001).

The key to tailoring these actions to different task contexts is acquiring the robotic equivalent of forward models: action models. These models predict for instance the performance of an action, or its expected success. Whereas forward models make their predictions on a motor control level, action models do so at the level of durative action. They perform their predictions on different time-scales. For example, the `goToPose` action takes the robot's current and goal pose, and when called continually, returns motor commands that will navigate the robot to the the goal pose. The action model that predicts the execution duration has the same signature, and predicts how long this navigation action will take till completion.

Because it is difficult and error-prone to manually specify action models, robots learn them from experience, gathered by executing the action and observing the result. This is Principle V from Section 1.1. These action models can be used to optimize action sequences, coordinate multiple robots, or adapt actions to new tasks. In this dissertation, actions are not merely fixed resources, they rather provide an initial innate 'action substrate'. Based on these innate actions, the robot learns more sophisticated actions and action parameterizations itself, by observing its actions, learning models of them, and using these models to tailor actions to new task contexts. With this approach, robots become more autonomous and adaptive.

The role of learning action models within the system has been highlighted in the system overview, depicted in Figure 4.1. For each action in the action library, one or more action models are learned. This is a two-step procedure, in which training data is first gathered by executing an action for random states from the post-condition, and transforming this data to an appropriate feature space. A generalized model is then learned from these examples by tree-based induction. This action model is than incorporated in the action for which it was learned, as shown in Figure 4.1.

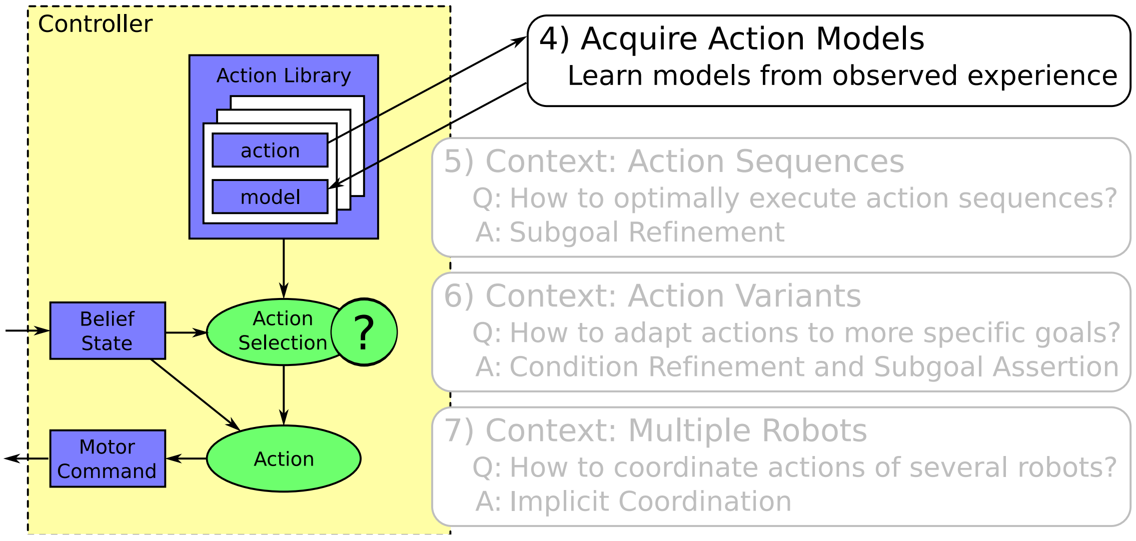

Controller
Action Library
action
model
Belief State
Action Selection
?
Motor Command
Action
4) Acquire Action Models
Learn models from observed experience
5) Context: Action Sequences
Q: How to optimally execute action sequences?
A: Subgoal Refinement
6) Context: Action Variants
Q: How to adapt actions to more specific goals?
A: Condition Refinement and Subgoal Assertion
7) Context: Multiple Robots
Q: How to coordinate actions of several robots?
A: Implicit Coordination

Figure 4.1. Acquiring action models within the overall system overview.

The next section presents how experience is gathered, and how this data is transformed to appropriate feature spaces. Section 4.2 presents an example of a learned action model. In Section 4.3, we evalute the learned models empirically. After discussing related work in Section 4.4, this chapter concludes with Section 4.5.

4.1 Acquisition of Training Data

Gathering training examples is done by executing an action, and observing the results. To ensure that an action can be executed, the initial and goal states are chosen from its pre- and post-conditions respectively. At the moment this is performed semi-automatically. The user defines ranges for the action parameters that ensure that the pre- and post-conditions are met, and the actual action parameters are sampled from these ranges randomly. The execution of an action from an initial to a goal state is called an episode. The procedure is as follows:

1. Choose a random initial and goal state from the valid range of action parameters. This ensure that the pre- and post-conditions are met, which guarantees that the action can be executed.

2. Select and execute another action that can achieve the initial state. For instance, if a model of the `dribbleBall` action is to be learned, the robot needs to be at the ball. If it is not, the `approachBall` action is executed beforehand. Using this preparatory action in experience gathering alleviates the need for human intervention with the robot, for instance, to make sure that the pre-conditions of an action are met. This substantially speeds up experience gathering in practice.

   Sometimes, this step can be bypassed. When the post-conditions of a action always satisfy its pre-conditions, e.g. when the pre-conditions are empty, the goal state of one action can be chosen to be the initial state of the next action, and there is no need for a preparatory action. Furthermore, in simulation, Step 2. can be eliminated by simply setting the state of the world to the initial state. Here, this instant environment modification can be seen as the preparatory action.

3. Execute the action for which a model will be learned, and record the observable state variables. Basically, all the variables in the robot's belief state are recorded. Which of them are relevant to learning the model is determined at a later stage. Realizing that an unrecorded variable might be relevant to learning the action model requires re-gathering the data, whereas recording all variables but not using all only costs memory. Most robots in this dissertation record their state at 10Hz, so an episode of s seconds duration contains $10s$ examples.

4. If enough examples have been gathered then quit, else repeat from Step 1. How much is "enough" is discussed in Section 4.1.3.

The running example in this section will be learning to predict execution time for the `goToPose` action for the simulated B21 in the kitchen environment. Figure 4.2 displays a concrete example of gathered training data with this robot. Here, 30 of 2948 executions of `goToPose` with random initial and goal states are shown.

2948 executions of `goToPose` takes approximately 7.5 hours to acquire. The total number of executions or episodes is called n_e. We split the data into a training and test set. The number

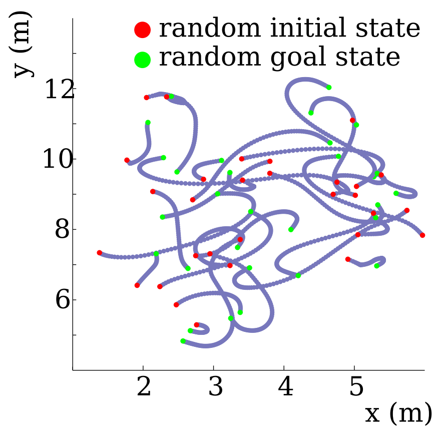


Figure 4.2. Experience for a the `goToPose` action (see Section A.1), in which the action was performed thirty times.

of examples in the training set is denoted N. If we include three fourth of the episodes in the training set, this yields $N{=}\frac{3}{4}n_e{=}2200$ examples. The question we now face is whether these 2200 examples are enough to train a good model? Will a learning algorithm trained with this amount of data likely make erroneous predictions on previously unseen cases? In general, any hypothesis that is consistent with a sufficiently large training set is deemed *probably approximately correct* PAC. A learning algorithm that has an error of at most ϵ with probability $1-\delta$ (i.e. is PAC) must be trained with at least N training examples, which can be computed with Equation 4.1.

$$N \geq \frac{1}{\epsilon}(ln\frac{1}{\delta} + ln|\mathbf{H}|) \tag{4.1}$$

Here, $|\mathbf{H}|$ is the number of possible hypotheses, which in our case are the model trees. Determining $|\mathbf{H}|$ for the model trees we use is beyond the scope of this research, but we can nevertheless use Equation 4.1 to determine how we can make the most of our data to learn accurate models. We use three approaches:

Reduce $|\mathbf{H}|$. 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 $|\mathbf{H}|$. This will be discussed in Section 4.1.1.

Increase N. Instead of using only the first initial example of each episode, we will also use intermediate data gathered on the way to the goal, as will be explained in Section 4.1.2. Here, we must be careful not to violate the stationarity assumption, which poses that the training and test set are taken from the same probability distribution.

Track ϵ empirically. By tracking ϵ over time as more data is gathered, we can determine when it stabilizes. At this point, we assume that $N \geq$ has been reached, and stop gathering data. We will demonstrate this in Section 4.1.3.

4.1.1 Appropriate feature spaces

Whilst gathering experience, the robot records all the observable and internal state variables. These might include the robot's pose, the ball's position, a teammate's position, and the target pose. Not all of these variables are relevant to learning an action model. For instance, if we are gathering experience for a navigation action, the position of the ball is irrelevant, whether it is seen or not. For learning, only relevant, also called informative (Haigh, 1998), features should be used.

Furthermore, the originally recorded state variables do not necessarily correlate well with the performance measure. Haigh (1998) calls such features *projective*. Finding feature spaces that correlate better facilitate learning, and less training examples are needed to learn an accurate model. The state variables recorded in the navigation task, shown to the left in Figure 4.3 are a good example. The original seven dimensional state space contains the initial and destination dynamic pose. The first column in Figure 4.3 shows these variables, along with a graph that plots the performance measure, time, as a function of one of these seven variables, x. The example points in these plots are the same as in Figure 4.2. x clearly does not correlate well with time, and neither do the other six features.

Fortunately, this state space contains several invariances, which can be exploited to derive feature spaces that correlate better with the performance measure. For instance, in the seven-dimensional state space, the learning algorithm has to learn to predict the execution duration for every initial and destination position separately. Of course, it is the relative position of the destination position to the initial one that matters, not their absolute positions. By exploiting this translational invariance, the state space can be reduced to the five-dimensional feature space depicted in the second column of Figure 4.3. Here, the robot is always at the location $(0, 0)$, and dx and dy are the difference between the x and y coordinates of the initial and destination position. A further reduction due to rotational invariance is possible, yielding the four-dimensional feature space depicted in the third column of 4.3.

By exploiting the invariances, we are reducing the dimensionality of the feature space. This again reduces the number of possible model trees which can be learned, which leads to a decrease of $|\mathbf{H}|$ in Equation 4.1. This equation also 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

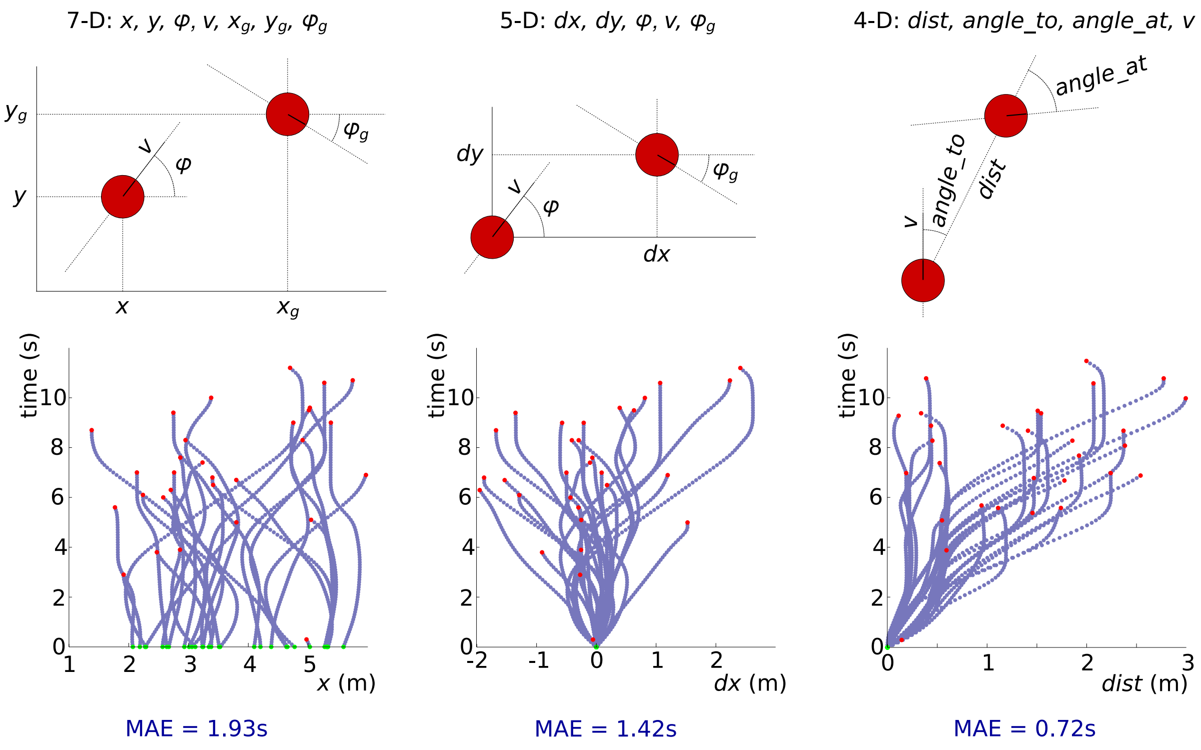


Figure 4.3. The original state space, and two derived feature spaces. The top figures depict the features used, and the graphs plot time against one of these features.

amount of data.

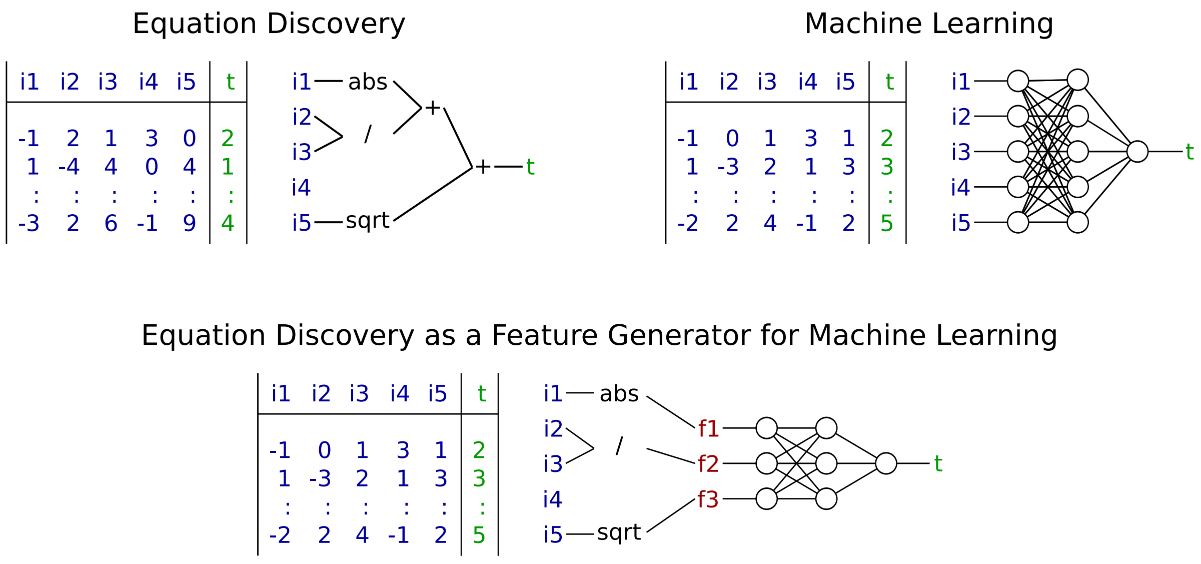
We have experimentally verified this, by training the model tree learning algorithm to be presented in Section 4.2 with data mapped to each of the three different feature spaces in Figure 4.3. For each feature space, the model was trained with 2200 (N) of the 2948 executed episodes. The Mean Absolute Error (MAE) of the each of these models was determined on the separate test containing the remaining $\frac{1}{3}N$ episodes. As can be seen in Figure 4.3, the MAE is lower dimensional feature spaces are used. We prefer the MAE over the Root Mean Square Error (RMSE), as it is more intuitive to understand, and the cost of a temporal prediction error is roughly proportional to the size of the error. There is no need to weight larger errors more.

Automatic feature space generation

For many applications, it is common to design feature spaces manually. State variables are composed into higher level features using domain-specific knowledge. Different learning problems often require different feature spaces. For instance, different actions might have different parameters and control different variables. Their action models will therefore need different features spaces to reduce $|\mathbf{H}|$, without abstracting away from relevant information.

Unfortunately, manually designing such feature languages is tedious, because each new learning problem usually needs its own customized feature space. It is also error-prone, as relevant information in the original state space might be lost in the transformation.

To overcome these problems, we propose an algorithm that automatically generates compact feature spaces, based on Equation Discovery (Stulp et al., 2006b). This is also known as Constructive Induction (Liu and Motoda, 1998; Bloedorn and Michalski, 1998). Equation Discovery systems introduce new variables from a set of arithmetical operators and functions. The algorithm explores the hypothesis space of all equations, restricted by heuristics and constraints. A classical representative is BACON (Langley et al., 1987), which rediscovered Kepler's law ($T^2 = kR^3$). A graphic example can be seen to the left in Figure 4.4, in which five input variables are mapped to the target by the equation $t = |i1| + (i2/i3) + \sqrt{i5}$. The advantage of Equation Discovery is that it yields a compact representation and human readable output. For instance, would the simplicity and elegance of Kepler's law be obvious from the learned weights in a neural network? However, the equations are restricted by the operators provided, and the hypothesis space that arises might not contain the true function. In these cases, the learning problem is said to be *unrealizable* (Russell and Norvig, 2003).



i1	i2	i3	i4	i5	t
-1	2	1	3	0	2
1	-4	4	0	4	1
:	:	:	:	:	:
-3	2	6	-1	9	4

i1	i2	i3	i4	i5	t
-1	0	1	3	1	2
1	-3	2	1	3	3
:	:	:	:	:	:
-2	2	4	-1	2	5

i1	i2	i3	i4	i5	t
-1	0	1	3	1	2
1	-3	2	1	3	3
:	:	:	:	:	:
-2	2	4	-1	2	5

Figure 4.4. Combining Equation Discovery and Machine Learning to generate features.

Our novel approach combines the strengths of Equation Discovery, being the compactness and interpretability of the resulting function, and Machine Learning, being its ability to approximate complex non-linear relationships. We do this by allowing Equation Discovery to discover many equations, which, when applied to the input data, yield data that has a higher correlation with the target data. Equation Discovery is halted at a certain depth, and from the multitude of generated equations (features), those most appropriate for learning are selected. The algorithm essentially searches for relationships between several input variables and the target variable that can be described well with operators, and leaves more complex relationships to machine learning.

The algorithm combines all l initial features with the k given operators, yielding new equations. These new features are added to the original set. This is repeated recursively d times, yielding equations with at most $2^{(d-1)}$ operators. Since the complexity of this algorithm is $\Theta(k^{2^d-1}l^{2^d})$, we should avoid generating irrelevant features. Mathematical constraints eliminate equations that generate neutral elements (e.g. x/x, x-x). Term reduction removes terms with the same semantics but different syntax (e.g. $x * 1/y = x/y$). Units are respected to avoid for example subtracting meters from millimeters, or meters from seconds. Furthermore, domain dependent operators can further control search. For example, in a geometrical domain it makes sense to add trigonometric operators and constraints how to use them, such as “apply $atan$ only to two distances”.

We further direct search by choosing only features that predict the target value well. This is done by computing the linear correlation coefficient r of the feature with the target value. This approach is fast, but suffers the same problems as other filter methods (John et al., 1994). At each depth, only the $p\%$ of features with highest correlation are added to the set for further processing.

Feature spaces for all action models

The feature spaces used to learn each of the models have been listed in Table 4.1. The formulae used to compute them from the action parameters listed in Table 2.1 are also given. The algorithm presented in the previous section was not used in all domains, but a preliminary version did find the appropriate features for the `goToPose` actions.

4.1.2 Including intermediate examples

To gather data, the initial and goal states for an action are chosen randomly from the range of valid action parameters. During execution, the observable and internal variables are recorded at 10Hz. These variables are then transformed into features. One such execution is called an episode. Part of an episode is depicted in Figure 4.5. For the running example, the B21 robot performed 2948 navigation actions, so this yields n_e=2948 episodes.

To train the learning algorithm, ideally only the first example of each episode should be

Robot	Action	Features
Pioneer I	`goToPose`	$dist = \sqrt{(x - x_g)^2 + (y - y_g)^2}$, $angle\_to = \lvert angle\_to_{signed} \rvert$, $angle\_at = sgn(angle\_to_{signed}) *$ $\quad norm(\phi_g - atan2(y_g - y, x_g - x))$
Ulm Sparrow	`goToPosition`	$v, dist, angle\_to$
B21	`goToPose`	$v_g, dist, angle\_to, angle\_at$, $\Delta angle = \lvert norm(\phi_g - \phi) \rvert$
	`reach`	$dist_{xyz} = \sqrt{(x - x_g)^2 + (y - y_g)^2 + (z - z_g)^2}$ $dist_{xy} = \sqrt{(x - x_g)^2 + (y - y_g)^2}, dist_{xz}, dist_{yz}$, $angle_{xy} = atan2(y_g - y, x_g - x), angle_{xz}, angle_{yz}$
PowerCube	`reach`	$dist = \sqrt{(\theta^a - \theta^a_g)^2 + (\theta^b - \theta^b_g)^2}$, $angle_1 = norm(atan2(y_g - y, x_g - x) - atan2(\dot{\theta}^b, \dot{\theta}^a))$, $angle_2 = norm(-atan2(y_g - y, x_g - x) + atan2(\dot{\theta}^b_g, \dot{\theta}^a_g))$ $v = \sqrt{\dot{\theta^a}^2 + \dot{\theta^b}^2}$ $v_g = \sqrt{\dot{\theta^a_g}^2 + \dot{\theta^b_g}^2}$

$norm(a)$: 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 4.1. The feature spaces used to learn action models

used. This is because only the first entries are from the same distribution as the distribution from which the initial and goal states were chosen. So if the original distribution from which these states are selected is uniform, the first entries will be uniformly distributed as well. This is necessary to fullfil the stationarity assumption, which demands that training and test set are taken from the same probability distribution (Russell and Norvig, 2003). This has been visualized in Figure 4.6, in the upper left graph. Here the initial states of thirty episodes are depicted, as in Figure 4.3. The distribution of the distance and time are shown in the histograms above and to the right of this graph. The histogram shows that initial distances to the goal are uniformly distributed. The model trained on these examples has a Mean Absolute Error of 0.59s.

From Equation 4.1, it can be derived that more training data (higher N) leads to more accurate models (ϵ) with higher probability (δ). For each episode, more data is easily acquired by using the execution duration not only from the initial state, but also from all the intermediate states to the goal. These extra examples are depicted in blue in Figure 4.5 and Figure 4.6, in the center upper graph. Instead of 2200 examples, we now have almost all 173336 examples, which is all the training data collected in almost 5 hours of action execution. Superficially, this

	time	v	v_g	$dist$	$angle\_to$	$angle\_at$
•	6.8	0.00	0.60	1.46	1.10	-1.63
•	6.7	0.00	0.60	1.46	1.10	-1.63
•	6.6	0.00	0.60	1.46	1.10	-1.63
:	:	:	:	:	:	:
•	3.5	0.53	0.60	0.65	0.98	1.23
•	3.4	0.51	0.60	0.62	1.02	1.16
•	3.3	0.48	0.60	0.60	1.05	1.08
:	:	:	:	:	:	:
•	0.2	0.40	0.60	0.08	0.03	-0.07
•	0.1	0.41	0.60	0.04	0.03	-0.06
•	0.0	0.43	0.60	0.00	0.00	-0.07

Figure 4.5. An example episode. The first entry is determined by the randomly chosen initial and goal state. The projective features in the final entry always pass through (0,0).

might seem the optimal choice: the maximum amount of data, and a lower error. However, a closer look shows another problem. Performance measures correlate with how far you are from the goal state. *How far* should be interpreted abstractly here; it could be a distance, an angle, some energy measure, time. Features that express well *how far* the robot is from the goal state, are usually good features for learning the model. Haigh (1998) calls such features *projective*. For instance, distance expresses very well how far we are from the goal, in a geometric sense.

Such measures are always defined relative to the goal position. The equation for computing distance ($\sqrt{(x - x_g)^2 + (y - y_g)^2}$) clearly shows that the first step is to subtract the goal coordinates from the current coordinates. Most features for learning action models compute their values relative to the goal state. This approach entails that when the goal is almost achieved, the distance measures will approach zero. The final row in the example episode in Figure 4.5 clearly demonstrates this. In the center graph of Figure 4.6 all episodes end in the origin at (0,0), even though the initial states are spread throughout the feature space.

The histograms around the center graph in Figure 4.6 show that both distance and time accumulate around zero. The distributions in the histograms are strongly skewed to zero. Similar patterns arise for other features and actions. The stationarity assumption is clearly violated. 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. Since it is more likely that the model will be inquired for states further from the goal, this is unacceptable.

One way to fullfill the stationarity assumption is to simply put all the intermediate examples from the episodes in the test set in the test set as well. Although both training and test set would then both be sampled from the same probability distribution, this distribution *does not* correspond to the distribution from which the goals were originally sampled. During real

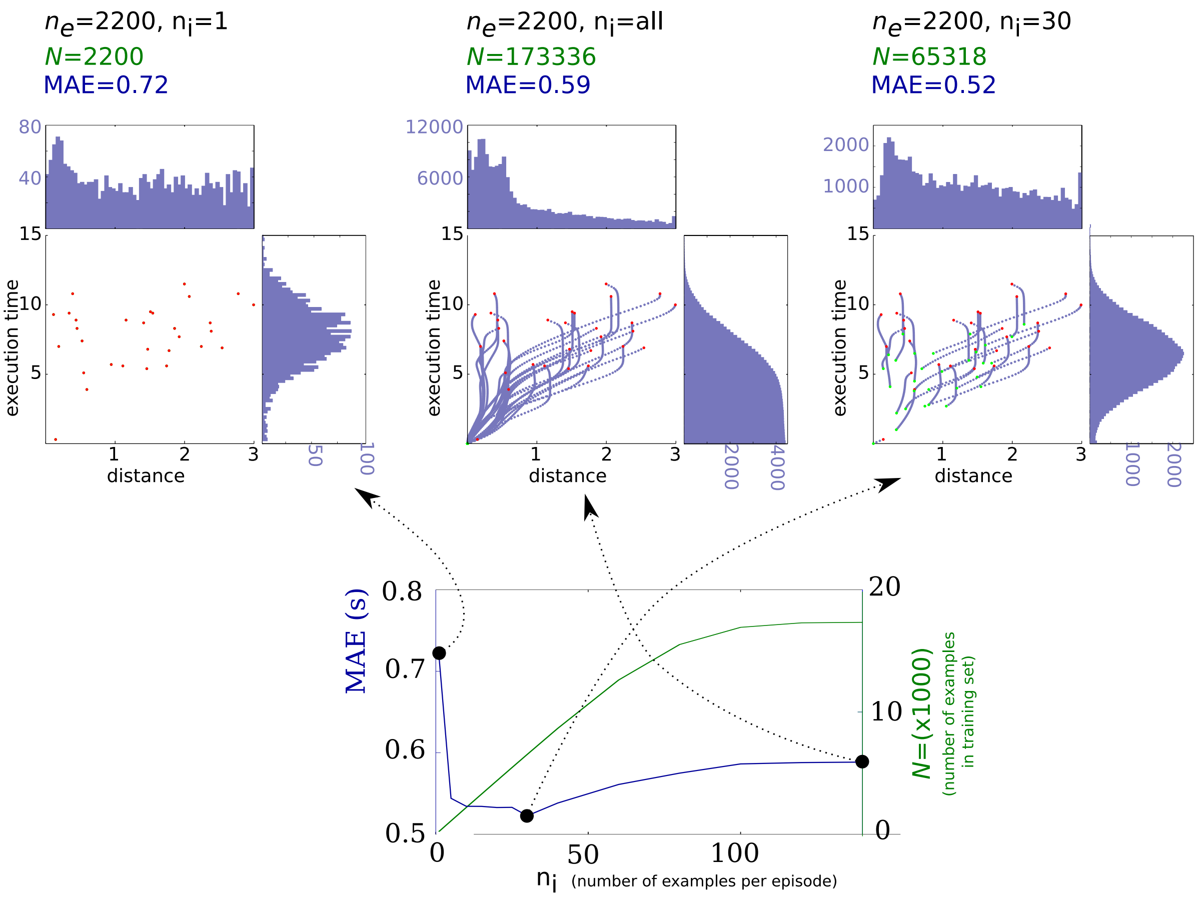


Figure 4.6. The lower graph depicts how the Mean Absolute Error and number of examples depend on the number of examples per episode used. The three upper graphs depict forty episodes, and the distribution of the examples for three values of examples/episode on the x-axis of the lower graph.

operation time the distances from the initial state to the goal will certainly not be as skewed towards 0 as in the center graph of Figure 4.6. However, it is exactly during operation time that we need the models to be accurate. Therefore, it is essential that our test set is sampled from the same distribution as during operatin time, which means we should only use the first example of each episode in the test set *and* fullfil the stationarity assumption when training the model.

A good compromise between the approaches of using only the first example or all examples of an episode is to use only the first few examples. The number of intermediate examples per episode included in the training data is denoted n_i. This means that the number of training examples is roughly $n_e * n_i$ 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 experimentally.

The lower graph in Figure 4.6 depicts how the Mean Absolute Error (MAE) of the learned model on a separate test set depends on n_i, the number of examples used per episode. In this case, the minimum value for MAE is 0.52s, when n_i is 30. This means the first 30 examples, equivalent to the first 3 seconds, of each episode are used. This yields a total of 65318 examples, as can be read from the right y-axis. Note that the number of examples grows linear with n_i at first, but settles at 173336 after a while. This is because none of the episodes has more than 139 examples (i.e. no episode took longer than 13.9s), so increasing the number of examples per episode has no effect. The upper left graph in Figure 4.6 shows these truncated episodes with n_i examples each, and the distribution of examples in the histograms. The distributions are close to the distributions from which the initial and goal states were sampled, shown in the right graph in Figure 4.6.

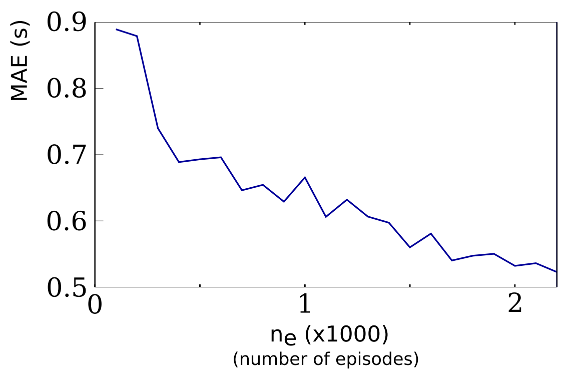
Summarizing, not all intermediate examples should be used to train an action model, as it the projective characteristics of good features will biases the model towards examples around the origin, thereby violating the stationarity assumption. On the other hand, using more data from each episode yields a accurate model. A compromise is to use the first n_i examples of each episode. The value of n_i that minimizes the MAE can be determined experimentally. Care must be taken not to evaluate the final model with the same test set that was used to determine n_i.

4.1.3 Number of training examples needed

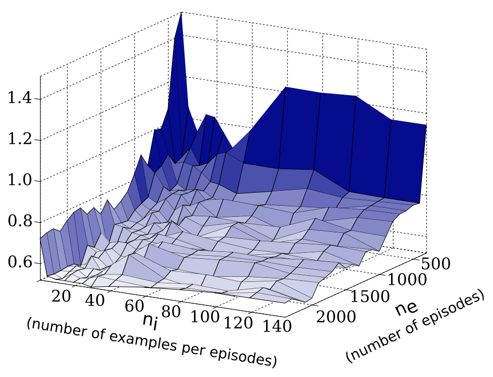
To learn an accurate action model, sufficient data must be available for the learning algorithm to build a model that generalizes well over unseen examples. On the other hand, the robot should not take days to collect its data. To analyze how many examples are needed to acquire an accurate prediction model, the model is frequently relearned as more and more examples become available. Once the mean absolute error between a separate test set and the prediction for these examples stabilizes, data acquisition is stopped.

Figure 4.7(a) demonstrates how the Mean Absolute Error (MAE) decreases as more episodes become available for training the model. Although the error has not stabilized completely, no more data is gathered. This is because the final model used on the robots is actually trained on all examples. Since there are no unbiased test examples left, its MAE cannot be determine, but this model can be expected to be more accurate than the model trained on the training set alone.

Finally, Figure 4.7(b) combines Figure 4.7(a) and 4.6 by showing the MAE for all combinations of n_i and the number of episodes available. There are two clear trends. First, more episodes means a more accurate model can be learned, which we had already concluded from



(a) The error of the learned model decreases as the number of episodes n_e increases.



(b) The error dependent on both n_e and n_i. The best value of n_i (30) is independent of n_e.

Figure 4.7. Collecting more examples lowers the model error.

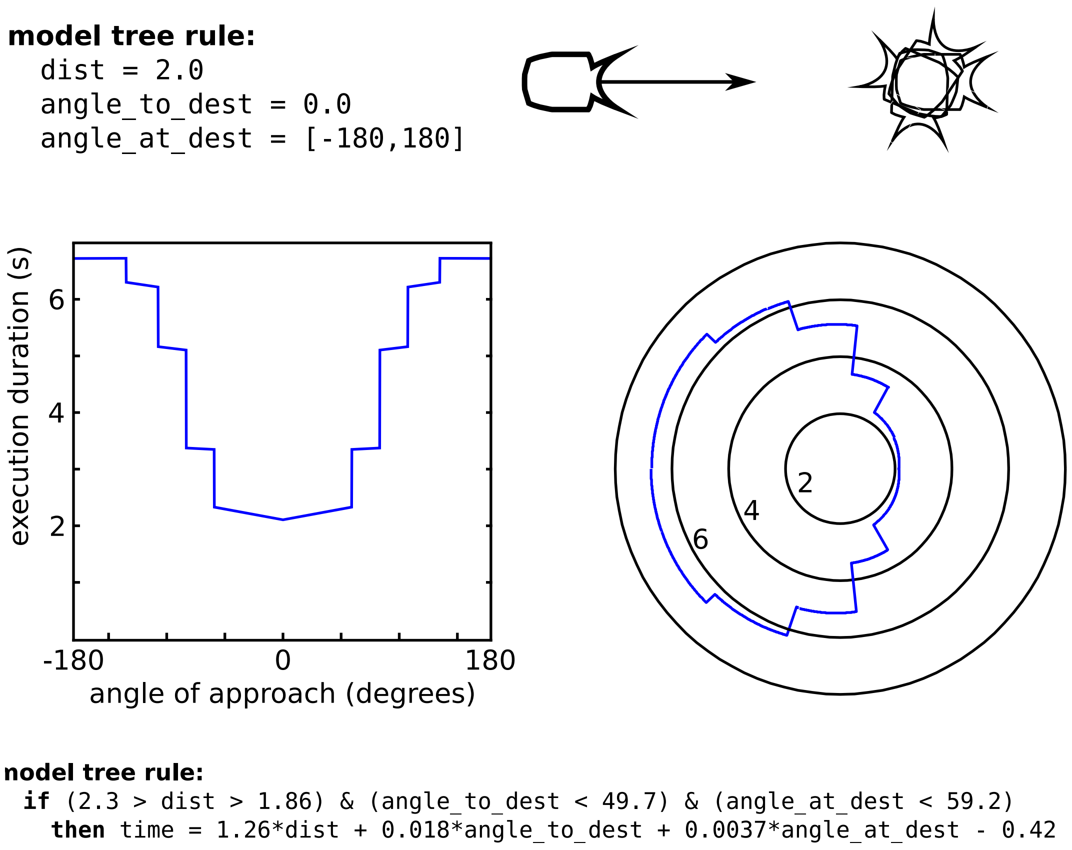
Equation 4.1, and visualized in Figure 4.7(a). Furthermore, the optimal value for n_i is largely independent of the number of episodes. It is actually also approximately the same across actions and domains. This means we do not need to redetermine n_i each time new data is gathered.

4.2 Learning Algorithms

Previous research on learning robot action models from observed experience has used neural networks (Buck et al., 2002b), as well as model trees (Balac, 2002; Belker, 2004) as learning algorithms. In (Stulp et al., 2006a), it was shown that there is no significant difference in the accuracy of action models learned with neural networks or model trees. However, decision and model trees have the benefit that they can be converted into sets of rules, which can be visually inspected. As we shall see in Section 5.3.2, model trees can be optimized analytically. Therefore, we will focus only on decision and model trees in this dissertation. Sections C.1 and C.2 in Appendix C describes these algorithms in more detail. Here we will present an example of a learned forward model.

In the soccer domain, the robots learn to predict the execution time of the `goToPose` action, described in Section A.1. The model was learned from 386 episodes. The first 20 examples per episode were used. The features used were $dist$, $angle\_to$, $angle\_at$ and v, see 4.3.

To demonstrate what the action models learned through model trees look like, an example of execution duration prediction for a specific situation is depicted in Figure 4.8. In this situation, the variables $dist$, $angle\_to$ and v (see Figure 4.3) are set to 2.0m, 0° and 0m/s respectively. The model is much more general, and predicts accurate values for any $dist$, $angle\_to$ and v. These variables are fixed for visualization purposes only. For these fixed values, Figure 4.8 shows how the predicted time depends on $angle\_at$, once in a Cartesian, once in a polar coordinate system.



```
model tree rule:
  dist = 2.0
  angle_to_dest = 0.0
  angle_at_dest = [-180,180]
```

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

Figure 4.8. 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 linear plot we can clearly see five line segments. This means that the model tree has partitioned the feature space for $dist$=2.0m $angle\_to$=0° and v=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. An arrow indicates its linear model in the plots. The polar plot clearly shows the dependency of predicted execution time on the angle of approach for the example situation. Approaching the goal at 0 degrees is fastest, and would take a predicted 2.1s. Approaching the goal at 180 degrees means the robot would have to navigate around the goal point, taking much longer (6.7s).

In Section 6.2, we will give a different example of an action model, which predicts if an action will succeed or fail. We have postponed this example, as it enables a clearer structuring of Chapter 6.

4.3 Empirical Evaluation

Table 4.2 lists the number of episodes N executed to gather data for the training set, the mean execution duration per episode $\bar{t}$, the total duration of data gathering for the training set $\bar{t} * N$, as well as the model's error (MAE) on a separate test set.

Robot		Action	$\frac{3}{4}n_e$	$\bar{t}$	$\bar{t} * \frac{3}{4}n_e$	MAE
				(s)	(h:mm)	(s)
Roboteq	R	`goToPose`	290	6.4	0:31	0.32
		`dribbleBall`	202	7.7	0:26	0.43
Pioneer I	R	`goToPose`	223	6.5	0:24	0.36
Pioneer I	S	`goToPose`	750	6.2	1:18	0.22
		`dribbleBall`	750	7.4	1:32	0.29
Ulm Sparrow	R	`goToPosition`	517	4.6	0:40	0.33
B21	S	`goToPose`	2200	9.0	5:45	0.52
		`reach`	2200	2.6	1:38	0.10
PowerCube	R	`reach`	1100	2.9	0:53	0.21

Table 4.2. This table lists the action models used in this dissertation.

For an unbiased evaluation of learned models, it is of course essential that the error measure is determined over a separate test, not the training set itself. The point of evaluation is to test how well the model generalizes over unseen examples. Sometimes, the test set is used to determine the parameterization of a learning algorithm. For instance, the learning algorithm is trained on the training set with different learning rates, and the learning rate which causes the lowest error on the test set is used for learning. We used this approach to determine n_i in Section 4.1.2. It is important to note that although the test set is not used to train the algorithm itself, it *was* used to train the learning parameter of the algorithm, and information from the test set has leaked into the resulting model. Therefore, we may not reuse this set for the final evaluation. This would be what Russell and Norvig (2003) consider *peeking*. The results in Table 4.2 have therefore been acquired as follows:

1. First, map the n_e episodes to the appropriate feature space
2. Then, the model tree is trained with $\frac{1}{2}n_e$ episodes for varying values of n_i. The best value of n_i is chosen based on the lowest error on a separate test set with $\frac{1}{4}n_e$ examples.

3. After determining n_i, the first test set is no longer need for testing, and it is added to the training set, which now contains $\frac{3}{4}n_e$ episodes, and approximately $N = \frac{3}{4}n_e n_i$ examples.
4. A model is trained with n_i intermediate examples per episode, and tested on the second test set, which contains the remaining $\frac{1}{4}n_e$ examples. The acquired error is reported in Table 4.2.
5. The final model stored in the action library was therefore trained with all n_e episodes, but could not be evaluated, as not test data is left. However, using more data should theoretically lead to a better model, according to Equation 4.1.

This might seem a bit cumbersome, but is essential to ensure that we do not peek, or use any training data to evaluate the learned model.

In the simulated domains and the PowerCube arm, data was gathered until the error stabilized. For the other first five actions, this was not yet the case. One reason is that gathering data on mobile robots is more cumbersome than in simulation or on fixed arms. The amount of data gathered for these actions has also consciously been kept low to demonstrate that good models can be learned in little time (e.g. <30 minutes). Even with limited data, and resulting sub-optimal accuracy of the action models, using these models for optimization and coordination still yields very good results, as we shall see in the next three chapters. In the conclusion in Section 4.5 we explain how more accurate models can be learned using data gathered on-line during robot deployment.

4.4 Related Work

Related work on learning forward models and actions models on robot has already been presented in Section 3.2.2 and 3.2.3. This sections will provide a comparison with the methods described in this chapter.

Most similar to our work is that of Belker (2004). Here, model trees are trained with data gathered from navigating through hallway environments. It was actually a discussion in exactly this hallway environment prompted us to use model trees, and extended their use to novel domains and actions.

Balac (2002) has developed the ERA (Exploration and Regression tree induction to produce Action models) system, in which robots learn the speed with which they can travel over terrains with different roughness properties, using regression trees. However, the speed with which a robot can navigate over different terrains could simply be acquired by navigating over the terrain and computing the mean speed, without using regression trees. A closer inspection of the visualized regression trees (see Balac et al., 2000, Figure 1) show that this is exactly what is happening.

Buck et al. (2002b) uses neural networks to learn execution duration prediction of a navigation action. These models are learned from data gathered during simulation, and have not been tested for accuracy on real robots. In this work, the number of examples needed, or the use of intermediate data is not investigated. We have found that neural networks and model trees do not have significant accuracy differences when trained on the same data to learn an action model (Stulp et al., 2006a).

Fox et al. (2006a) propose the use of Hidden Markov Models to learn action models. As this work has more relevance to Chapter 6, it will be explained more elaborately in Section 6.5.2.

4.4.1 Reinforcement Learning

The important aspect of Reinforcement Learning for this dissertation is that V- and Q-values can be thought of as action models. In this case the action is the optimal policy, and the value is the predicted reward at some future time. The main difference between V- and Q-values, is that action models are:

Reusable V- and Q-values are learned specifically for a certain environment, with a specific reward function representing a specific goal. The values are then learned for all states, but for a single goal. 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.

Meaningful The performance measures we can learn, such as execution duration, are informative values, with a meaning in the physical world. Rewards have no unit, and are chosen arbitrarily.

Composable Because action models return meaningful values, these values can be composed into more complex values. For instance, a composed performance measure could take both execution duration and energy consumption into account. Since the Value compiles all performance information in a single non-decomposable numeric value, it cannot be reasoned about in this fashion.

Modular In Hierarchical Reinforcement Learning, Q-values are learned in the calling context of the action. Policy optimization can therefore only be done in the context of the pre-specified hierarchy/program. Action prediction models are independent of the calling context, so can be combined in any order. Also, the scale of rewards are determined arbitrarily. They can be 1000 or 1. Therefore, it is not possible to add the rewards or values of two actions in a meaningful way, for instance if a sequence of actions is considered. Maybe one has received a reward of 1000 for successful execution, and the other only 1. These two aspects prevent Reinforcement Learning systems from being able to optimize action sequences that have been generated on-line, such as in planning. Ryan and Pendrith (1998) proposes RL-TOPS (*Reinforcement Learning - Teleo Operators*), the only approach we know of that explicitly combines planning and Reinforcement Learning is. This work will be described in more detail in Section 5.5

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

The benefit of Reinforcement Learning algorithms is the rigorous mathematical framework they provide, along with extensive experimental research on improving the algorithms.

4.5 Conclusion

In cognitive science, motor prediction is the key solution to many of the problems encountered in motor control. Predicting the outcome of actions is learned from observed experience. In this chapter, we have described a similar process for robots. The first step is to acquire experience by simply executing the action. This state space of this data is mapped to a feature space with lower dimensionality, so less data is needed to learn an accurate model. Intermediate data between the start and end of an episode is be included, whilst taking care that the stationarity assumption is not violated, which is bound to happen due to the projective nature of good features. Data acquisition is stopped when the error of the learned model stabilizes. It was demonstrated that accurate action models could be learned for the actions of several simulated and real robots.

A benefit of using model trees is that they tend to only use variables that are relevant to predicting the target value. Initial learned models with relatively few relevant features could be learnt quite accurately, but we expect that more complex actions with many degrees of freedom, and therefore a large number of parameters, can not be learned as accurately with the amount of data gathered on-line. We propose three solutions for this problem:

- Evaluate other function approximators that deal well with high dimensional search spaces.

- Further investigate the use feature space discovery systems, such as the one described in Section 4.1.1, that can find useful abstractions for learning.
- Train the models that is gathered with data on-line during operation time.

Especially this last measure will likely have a very positive impact on the accuracy of the learned models. First of all, since data gathering is then not done off-line, but parallel to actual robot deployment, more training data can be acquired. More importantly though, the training data contains action parameterizations that are not generated randomly, but rather arise during actual operation. In general, it is to be expected that future experiences will be similar to past experiences. Therefore, the training set (experiences from past actions) will be from the same probability distribution as the 'test set' (future experience from yet to be executed actions). In a sense, we could therefore say that the stationarity assumption is fullfilled with respect to future unseen actions. The stationarity assumption is necessary to guarantee that the learned model is probably approximately correct with respect to unseen examples (Russell and Norvig, 2003). **?**) has demonstrated empirically in the service robotics domain that training models with data gathered on-line improves the action model accuracy during operation.

We could image that robots operating in a variety of real world environments could first be provided with default general action models learned from uniformly distributed examples. When the robot is put into operation, it starts gathering data itself, and retrains the action models with this data. It is to be expected that the models so obtained will be tailored to the context the robot is acting in, and therefore more accurate in than the general default action model.

We have also done some preliminary work on learning the effects of an action on a parameter level. For instance, it might be able to parameterize an action with a target location x_g, y_g, but this location will not be perfectly reached. By comparing the true final location with the target parameters, a robot can learn the accuracy and robustness of an action. This could enable the robot to make well-informed decisions on how to parameterize an action. For instance, the learned models showed that a high target translational velocity causes the robot to reach the target less precisely. If a target needs to be reached with high precision, the robot could choose to select a lower translational velocity.

The results reported in this chapter have been published in: (Stulp and Beetz, 2005b,c,a, 2006; Stulp et al., 2006a; Isik et al., 2006; Stulp et al., 2006b, 2007). Summaries of these publications are given in Appendix D.

5. Task Context: Action Sequences

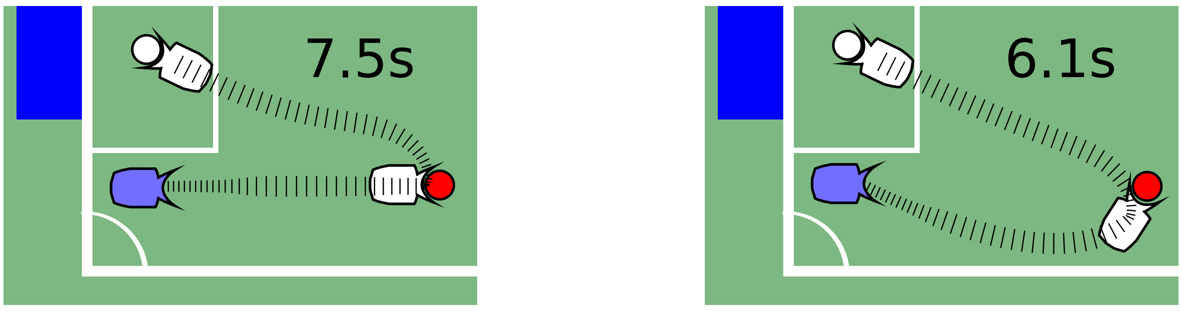
"It seemed to Quinn that Stillman's body had not been used for a long time and that all its functions had been relearned, so that motion had become a conscious process, each movement broken down into its submovements, with the result that all flow and spontaneity had been lost."

Paul Auster – The New York Trilogy

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. For instance, there is a dance called "The Robot" which, according to Wikipedia is characterized by *"...all movements are started and finished with a small jerk..."*. Auster (1987) gives an accurate description of this type of motion when introducing the character Stillman, a seriously ill person, in the quote above.

In contrast, one of the impressive capabilities of animals and humans is their capability to perform sequences of actions efficiently, and with seamless transitions between subsequent actions. As was mentioned in Section 3.2.1, there are often infinitely many ways to perform any task, but most tasks are solved with highly stereotyped and smooth movement patterns (Wolpert and Ghahramani, 2000). It is assumed that that these typical patterns are those that minimize a certain cost function. So, in nature, fluency of motion is not a goal in itself, but rather an emergent property of time, energy and accuracy optimization. In cognitive science, one challenge is to reverse-engineer this cost function, given the motion patterns found in empirical studies. In this section, we invert the process, and demonstrate that requiring optimal execution of action sequences with respect to time also automatically leads to smooth natural motion in robots.

Figure 1.2, repeated in Figure 5.1, demonstrates an abrupt transition when approaching the ball to dribble it to a certain location. As discussed in Section 1.1, Principle III, these abrupt transitions often arise because action abstractions abstract away from aspects that influence the performance. In this case, the angle of approach is abstracted away from when selecting the actions, although it obviously influences the execution duration.



(a) An execution with an abrupt transition at the intermediate goal.

(b) An time-optimal execution that exhibits smooth motion.

Figure 5.1. A greedy and an optimal execution of the same abstract action sequence.

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. Because the angle of approach is not fixed, many intermediate subgoals are possible. Automatically determining the optimal intermediate subgoal is called *subgoal refinement*. It is based on extracting and optimizing *free action parameters*. The optimal values of free action parameters are determined by requiring the expected cost of the execution of the entire sequence of actions to be as small as possible. In the example above, the free action parameter was the angle of approach, and the expected cost is time, which can be predicted with action models described in Chapter 4.

The behavior shown after applying subgoal refinement in Figure 1.2(b) has a higher performance, achieving the ultimate goal in less time. A pleasing side-effect is that it exhibits seamless transitions between actions. The plots of the navigation trajectories in the fields demonstrate this. The lines on the trajectories represent the robot's pose and translational velocity, recorded at 10Hz. The center of each line is the robot's position. The lines are drawn perpendicular to the robot's orientation, and their width represents the translational velocity at that point.

The main motivation for subgoal refinement from a controller design point of view is that human designers or planning systems should reason only about abstractions of actions (Principle I), and have the robot automatically optimize aspects of the action that are relevant for its execution with subgoal refinement (Principle IV).

In Figure 5.2, subgoal refinement has been highlighted within the system overview. The subgoal refinement module takes an action sequence as its input, possibly with free action parameters, and returns the same action sequence, with refined subgoals.

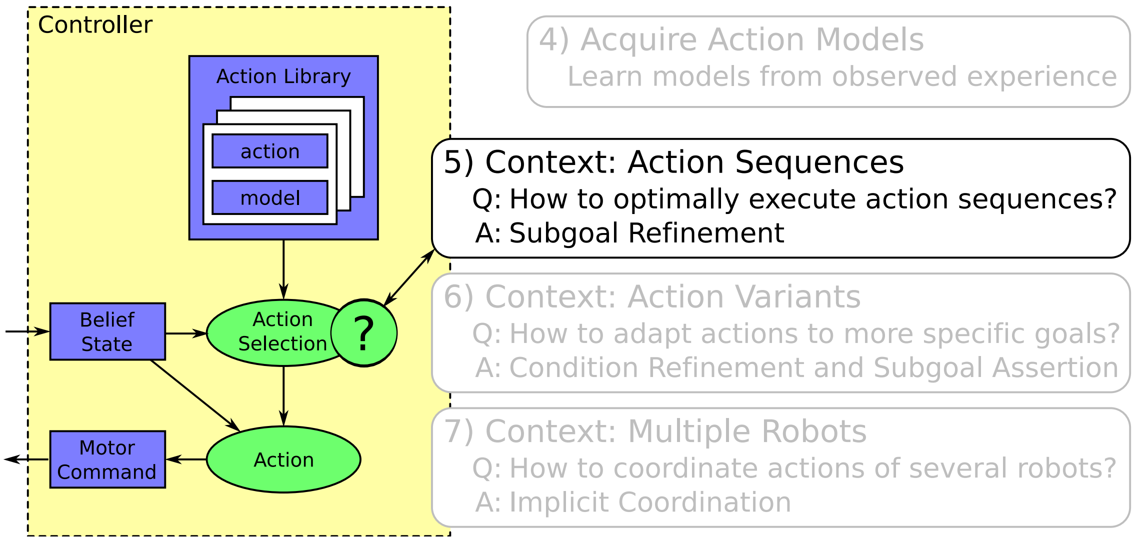

Controller
Action Library
action
model
Belief State
Action Selection
?
Motor Command
Action
4) Acquire Action Models
Learn models from observed experience
5) Context: Action Sequences
Q: How to optimally execute action sequences?
A: Subgoal Refinement
6) Context: Action Variants
Q: How to adapt actions to more specific goals?
A: Condition Refinement and Subgoal Assertion
7) Context: Multiple Robots
Q: How to coordinate actions of several robots?
A: Implicit Coordination

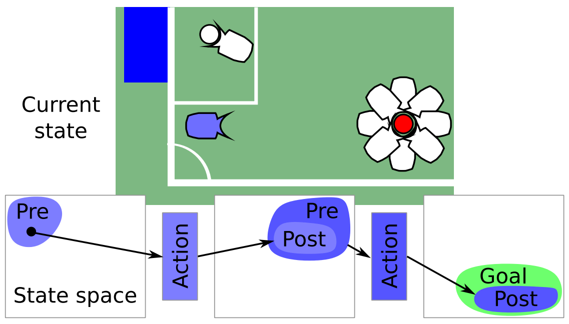
Figure 5.2. Subgoal refinement within the overall system overview.

The rest of this chapter is organized as follows. In the next section, the computational model for subgoal refinement is introduced. The process of generating abstract action sequences through planning is presented in Section 5.2. The implementation of subgoal refinement, the procedure of extracting and optimizing free action parameters, is introduced in Section 5.3. An empirical evaluation of the effects of subgoal refinement in the three robotic domains is presented in Section 5.4. Related work is discussed in Section 5.4, after which we concluded with Section 5.6.

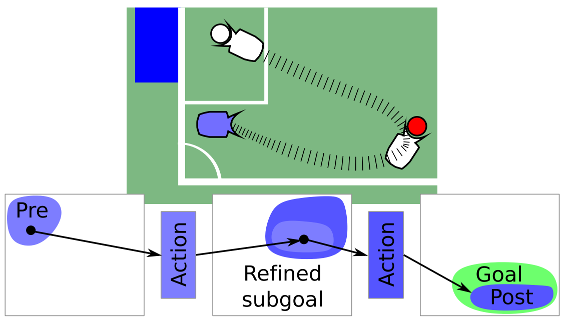
5.1 Computational Model

Subgoal refinement can best be explained in the context of abstract action chains. In an abstract action chain, the postconditions of each action satisfy the preconditions of the next action. Preconditions of an action constrain the possible states in which the action can be executed, and the postconditions the states that might arise when executing the action until completion. Figure 5.3(a) depicts an abstract action chain, with pre- and postconditions represented as subsets of the entire state space.

Note that there are many possible intermediate states, as the intersection of pre- and postconditions yields a whole set of possible states, not just one. In the ball approach example,



(a) Abstract action chain before subgoal refinement.



(b) Abstract action chain optimized with subgoal refinement.

Figure 5.3. Computational model of subgoal refinement.

this set of intermediate states contains all possible states in which the robot is at the ball, eight of which are also depicted in Figure 5.3(a). In this set, all variables are equal, except the angle with which the ball is approached. This action parameter is therefore called *free*. The first step in subgoal refinement is automatically determining the free action parameters in a sequence of abstract actions, by examining their pre- and postconditions.

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 5.1 some values are better than others, with respect to the expected performance. Therefore, the second step in subgoal refinement is to choose values for the free action parameters that minimize the expected cost of executing the entire sequence of actions. The expected cost is predicted using action models.

To optimize action sequences, the robot must first generate action sequences. In this dissertation, this is performed using a symbolic planner. The general computational model of symbolic plan-based robot control is depicted in Figure 5.4, and is similar to the models proposed by Bouguerra and Karlsson (2005) and Cambon et al. (2004), which will be discussed more detail in Section 5.5.3.

The first step is to convert the continuous state variables in the belief state to an abstract state, through a process called anchoring. Given this abstract state, a goal, and the action library, the planning system then generates a chain of abstract actions that can achieve the goal. The abstract actions in this plan are then instantiated, given the corresponding executable actions in the action library, and the state variables in the belief state. Subgoal refinement takes the (partially) instantiated action sequence, and optimizes it. Note that subgoal refinement module does not interact at all with the planning or execution processes, but only modifies

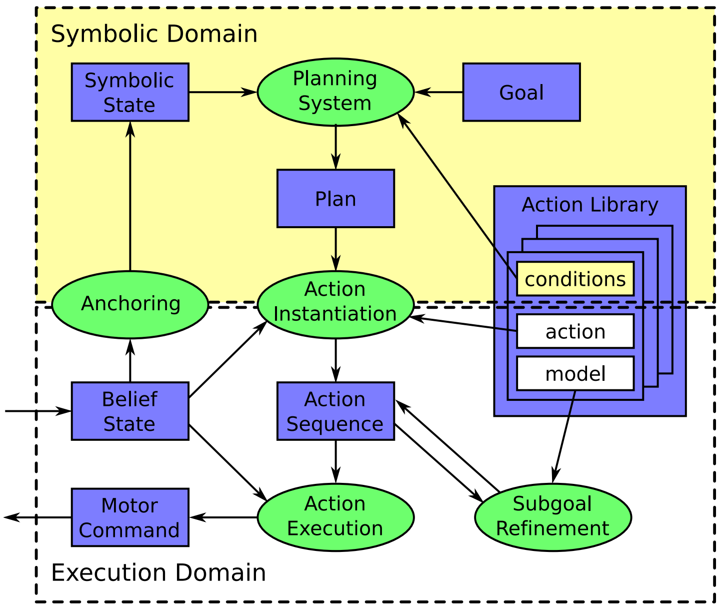


Figure 5.4. Computational model of subgoal refinement in action sequence generation.

existing action sequences.

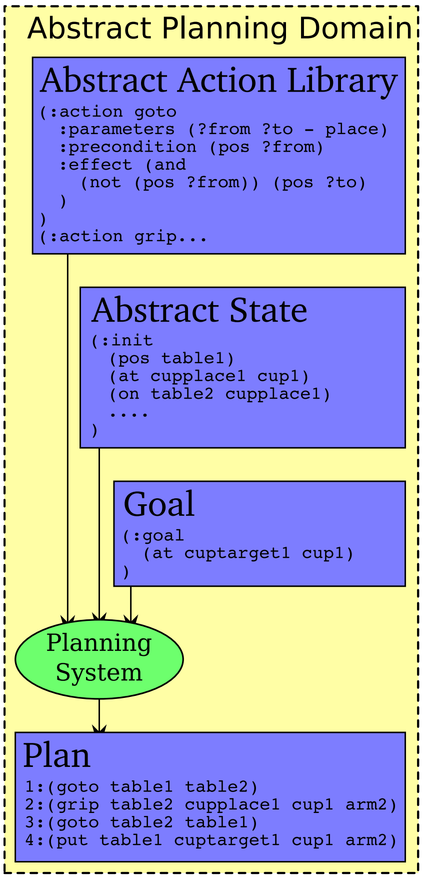
5.2 Action Sequence Generation

The next sections will present the implementation of the anchoring, planning system and action instantiation from Figure 5.4. We also describe how abstract states, goals and plans are represented. In Section 5.3, the implementation of subgoal refinement is introduced.

5.2.1 Abstract actions: PDDL

Whereas the executable action specifies *how* the action performs it task, the abstract action specifies *what* the action can perform. To be more precise the preconditions of an action define in which states the action can be applied successfully, and the effects (or postconditions) define what the effect of continually executing the action till completion will be.

In the system implementation, the Planning Domain Description Language (PDDL2.1 (Fox and Long, 2003)) is used to describe abstract actions, abstract states and goals. The benefit of using this language is that it is used as the input and output format of the International Planning Competition, held biannually in conjunction with International Conference on Automated Planning and Scheduling, making it a standard in the planning community. For this reason, there are many tutorials and examples available for PDDL, as well as a multitude of planning system implementations that efficiently generate PDDL plans.



```
(:action goto
  :parameters (?from ?to - place)
  :precondition (pos ?from)
  :effect (and
    (not (pos ?from)) (pos ?to)
  )
)
(:action grip...
```

```
(:init
  (pos table1)
  (at cupplace1 cup1)
  (on table2 cupplace1)
  ....
)
```

```
(:goal
  (at cuptarget1 cup1)
)
```

```
1:(goto table1 table2)
2:(grip table2 cupplace1 cup1 arm2)
3:(goto table2 table1)
4:(put table1 cuptarget1 cup1 arm2)
```

Figure 5.5. PDDL planning example

The actions in the action library, along with their preconditions and effects are specified in PDDL, as depicted in the example from the service robotics domain in Figure 5.5. The effects contains an add-list and a delete-list, that specify which new facts should be added and removed to the abstract state. As can be seen, actions and their conditions are represented by easy to interpret symbols.

Figure 5.5 depicts examples of an initial state and a goal in the service robotics domain. Due to the symbolic nature of PDDL, these specifications are on a level of abstraction that can be understood by humans who have no experience with PDDL, or planning in general. In this dissertation, goals are specified manually, depending on the scenario, as is done in the International Planning Competition. In the context of a full robotic controller, rules that determine goals online could be written.

The output of a PDDL planner is a list of abstract action with symbolic parameters, also depicted in Figure 5.5. In a chain of abstract actions the precondition of the first action is satisfied by the current situation, and the preconditions of all other actions are satisfied by the postconditions of preceding actions. The postcondition of the last action must satisfy the goal. A chain of abstract actions represent a valid plan to achieve the goal. Each action essentially enables the next action to be executed, until the goal is reached. Note that an action sequence is a list of executable actions with (partially) instantiated usually continuous parameters. They are called sequences rather than chains, to emphasize that the strong causal link between subsequent abstract actions in a chain is not explicit in action sequences.

Converting the continuous variables from the belief state into named symbols (e.g. PDDL symbols) is called anchoring (Coradeschi and Saffiotti, 2001). As we currently do not consider replanning, anchoring need only take place at the beginning of the planning process. As anchoring is not the focus of this research, we manually specify the initial abstract state, which is constant for each scenario presented in Section 5.4. These limitations will be discussed in more detail in Section 5.2.3. The actual planning process used to generate PDDL plans from PDDL action and state specifications is performed by the Versatile Heuristic Partial Order Planner (Younes and Simmons, 2003)[1].

5.2.2 Action instantiation

The PDDL plans that VHPOP generates are very abstract, with clear semantics of *what* actions do, even without knowing how the actions are executed. This makes human inspection of the plan feasible. However, it does not specify *how* this plan can or should be executed in the real world. Therefore, once the plan is acquired, the abstract action sequences must be instantiated with the corresponding executable actions. For instance, the abstract action `(goto start ball)` is converted to an action by determining the locations of `start` and `ball` by determining the coordinates of these objects in the belief state, and instantiating the appropriate action with them.

Let use suppose the soccer robot has a simple action `goToPosition`, that takes it from a certain position `(x,y)` to a goal position `(xg,yg)`. Given the symbolic plan `(goto start ball)`, the `goToPosition` action can be instantiated by determining the `(x,y)` positions of the ball, and passing these as parameters to `goToPosition`, as in Figure 5.6. Since all the parameters of the action are bound to values, this action is fully instantiated. This process is called *operator instantiation* (Schmill et al., 2000).

```
Action instantiateAction(abstract_action, params[], belief_state) {
  if (abstract_action == "goto") {
    x = belief_state.getX(params[0])  // params[0] => "start"
    y = belief_state.getY(params[0])
    xg = belief_state.getX(params[1]) // params[1] => "ball"
    yg = belief_state.getY(params[1])
    return GoToPositionAction(x,y,xg,yg)
  } else if ...  (other actions)

instantiateAction("goto", ["start", "ball"], belief_state) =>
   goToPosition( x=2.68,  y=2.12, xg=4.17, yg=3.40 )
```

Figure 5.6. Fully instantiated action.

5.2.3 Discussion

Using symbolic planners to generate action sequences for robots has a long tradition. Shakey, one of the first autonomous mobile robots used PDDL-style representations to determine ac-

[1]This planner can be downloaded free of cost at `http://www.tempastic.org/vhpop/`

tion sequences that would achieve its goal (Nilsson, 1984; Fikes and Nilsson, 1971). More recent examples include the work of Coradeschi and Saffiotti (2001), Cambon et al. (2004) and Bouguerra and Karlsson (2005). The approach explained in this chapter contributes to this research area. Some reasons why symbolic planning is of interest to robotics are:

Abstraction Symbolic planners abstract away from many aspects of the belief state, so planning and replanning is faster and more complex problems can be dealt with.

Adaptation Action sequences or action hierarchies must not be specified in advance, but are generated on-line, depending on the situation at hand. This makes the system more adaptive. The designer need only specify the pre- and postconditions of an action, independent of the other actions in the library.

Predictive plan repair Robot can reason about plans off-line before execution, to recognize and repair failures (Beetz, 2000) in advance. Of course, this is preferable to encountering them during task execution.

Constraints Constraints on actions can be specified symbolically. Cambon et al. (2004) uses symbolic constraints to intuitively specify that larger objects cannot be placed upon smaller ones.

VHPOP is, as most PDDL planners, a general purpose planner, not specifically tailored to robot planning. It does not address some of the problems inherent in robotics. The system presented in this section abstracts away from these problem to focus on the main contribution: the optimization of already generated plans. Other research has focussed on other problems that need to be resolved to enable symbolic planning on robotics, such as uncertainty, failure recovery and action monitoring ((Bouguerra and Karlsson, 2005)), geometric constraints ((Cambon et al., 2004)), and anchoring ((Coradeschi and Saffiotti, 2001)).

Uncertainty The symbols used in the symbolic state are either true or not. In robotics applications, this certainty cannot be achieved. The system would be more robust if it took uncertainty into account. Bouguerra and Karlsson (2005) present a system in which probabilistic representation of states and a probabilistic planner are used.

Geometric constraints In robotics, the robot and objects physically take up space in the world. This places geometric constraints on the movements the robot can make, and the interactions that are possible with objects. The aSyMov (Cambon et al., 2004) system takes these constraints into account, and maps them to preconditions for actions.

Failure recovery The current version of our system does consider failure recovery or replanning. In robotics, action can or are not always executed, and their desired effects not achieved. This requires that the plan is repaired or replanned from scratch. Bouguerra and Karlsson (2005) propose a method for monitoring action execution, and recognizing action failure.

Anchoring Anchoring usually involves complex tracking mechanisms to maintain the correspondence between symbols in the symbolic state, and objects locations in the belief state. Coradeschi and Saffiotti (2001) provide an overview of anchoring in robotic planning.

Implicit abstract representations

In Section 3.1.1, direct programming as a method to manually design controllers was introduced. In this approach, the abstract planning domain in Figure 5.4 is not explicitly represented in the controller. However, it is implicitly represented *in the designer's mind*. Consider the following, trivial hand-coded soccer playing controller in Figure 5.7.

```
...
  if (hasBall(beliefState)) {
    if (facingGoal(beliefState)) {
      shoot(beliefState)
    } else {
      dribbleToGoal(beliefState)
    }
  } else {
    approachBall(beliefState)
  }
...
```

Figure 5.7. Hand-coded soccer action selection module.

This code has no merit in itself, except demonstrating how following abstract concepts are represented implicitly:

Sequentiality the control flow of the program will ensure that the action sequence `approachBall` - `dribbleToGoal` - `shoot` is executed. This sequence of actions are not known in advance, but rather arise implicitly by traversing through state space, thereby also traversing the corresponding action space.

Abstract State and Action the function `hasBall` abstracts away from many aspects of the state, and compresses it into one boolean value. `hasBall` also implicitly encodes the precondition of both `dribbleToGoal` and `shoot`.

Abstract Goal From this code alone it is clear to us that the robot's purpose is to score a goal.

In principle, subgoal refinement can also be implemented without a planning system or explicitly encoding conditions. If there is only a fixed number of action sequences, the designer can still enable subgoal refinement by explicitly specifying the free action parameters and the models with which respect they should be optimized for each action transition. This is actually how the subgoal refinement system was initially implemented, before the planner was added.

Purely reactive systems cannot use subgoal refinement, as it depends on the commitment to a future sequence of actions. If it is not clear that the ball will be dribbled after having approached it, the robot cannot anticipate the best angle to approach the ball at. Both direct programming (Section 3.1.1) and motion blending (Section 3.1.2) methods often use hysteris to avoid too frequent switching between behaviors, and the unfluent motion that arises (Lötzsch et al., 2004; Kobialka and Jaeger, 2003). Note that hysteris is essentially committing to an action for a certain amount of time. Apparently, even reactive systems cannot dispense of commitment completely to avoid jagged motion.

We believe that explicitly encoding action abstractions is preferable, as having knowledge about your own actions enables the robot to reason about and manipulate them itself. This is essential for autonomy, adaptivity, and intelligent behavior in general (Dearden and Demiris, 2005). For instance, it allows subgoal refinement to be automated, and applied to previously unknown action sequences.

5.3 Subgoal Refinement

Now that the action sequences have been generated, they can be optimized with subgoal refinement. Subgoal refinement is based on the concept of free action parameters. This section will first describe what free action parameters are, and then how they are automatically extracted and optimized.

5.3.1 Free action parameters

Contrary to classical AI planning, actions are often redundant and over-expressive, both in nature and robotics applications. In robotic arm control for instance, one gripper position can often be achieved by many joint configurations (Hooper, 1994). In nature, there are many ways that a predator can stalk its prey, and some will work better than others (Blumberg, 2003). For instance, when stalking their prey, wild cats press themselves to the ground to avoid being seen. Apparently, this is much better than walking up-right. Domesticated cats instinctively crouch too, but, in over-enthusiastic anticipation of a prey never to be caught, tend to vigorously wave their tail in the air at the same time. Their survival does not depend on actually catching prey, and thus their stalking skills have never been perfected. Although at an abstract level, the same action is being executed, it is just a small variation that determines success or failure, to the frustration of domesticated cats world-wide.

The free action parameters in robotic actions are simultaneously a blessing and a curse. Although modern robot planners do not make the assumptions that classical planners do, planners still view actions at a level of abstraction that ignores the subtle differences between actions. Note that this is a blessing, because such details should not be considered at the abstract planning level, to keep planning tractable and preserve its declarative nature.

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 contexts of their execution. This curse often yields suboptimal behavior with abrupt transitions between actions, as we saw in the example in Figure 5.1(a). 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 these variables are indeed irrelevant to the validity of the plan, they are relevant to the performance of plan execution. Our system allows planners to reason about high-level abstractions of actions, but also optimizes the way in which the action is performed at a lower level. The curse becomes a additional blessing.

We will now explain how free action parameters are determined in the action instantiation process in the planning system. In Figure 5.6, we saw how the `goToPosition` action was fully instantiated by the abstract plan `(goto start ball)`. Now, instead of instantiating the `goToPosition` action, we will instantiate the `goToPose` action with the same plan. The `goToPose` action also contains the current and goal orientation and velocity. Since the symbol `ball` does not have an orientation in the belief state, this leads to free parameters.

These free unspecified parameters are usually set to default values, such as ϕ_g to 0.0 in the example in Figure 5.1. However, since this value is *unspecified*, it is more sensible to set it to the range of all possible values it can take. Figure 5.8 shows how unspecified parameters are set to default ranges in the program.

This program clearly demonstrates the parameters which the abstract plan has abstracted away from. Note that the unspecified values do not influence the validity of the plan, as they are not related to the symbolic parameters of the abstract plan.

```
Action instantiateAction(abstract_action, params[], belief_state) {
 if (abstract_action == "goto") {
   x = belief_state.getX(params[0])
   y = belief_state.getY(params[0])
   phi = belief_state.getPhi(params[0])
   v = belief_state.getV(params[0])
   xg = belief_state.getX(params[1])
   yg = belief_state.getY(params[1])
   phig = [-PI,PI]      // phig not specified, set to default
   vg = [VMIN,VMAX]     // vg not specified, set to default
   return GoToPoseAction(x,y,phi,v, xg,yg,phig,vg)
 } else if ... (other actions)

instantiateAction("goto", ["start", "ball"], belief_state) =>
  goToPose(  x=2.68,  y=2.12, phi=0.21, v=0.4,
            xg=4.17, yg=3.40, phig=[-PI,PI], vg=[VMIN,VMAX] )
```

Figure 5.8. An action sequence with free parameters set to default ranges.

In real scenarios, the default ranges of parameters are often constrained further by the context the action is being executed in. For instance, the maximum velocity of the soccer robots is 2m/s, but the precondition of `dribbleBall` specifies that it should be in the range [0m/s,0.3m/s] because the robot will not be able to control the ball otherwise. So the goal velocity of a `dribbleBall` action will always be in this range, and the starting velocity should be in this range too. Figure 5.9 demonstrates this approach as an extension of the previous two programs. Instead of setting the free action parameters to default ranges, they are set to ranges that depend upon the context, which is passed as an extra parameter.

In partial order planning, a causal link is a pair of actions and a proposition, which is a postcondition of the first action, and the precondition of the next. In Figure 5.9, the designer of the program is essentially reasoning about such causal links between actions. Will the ball be approached *in order* to dribble or shoot it? These different causes or contexts yield different constraints on the free action parameters, such as the more restricted velocity for ball dribbling.

Although the program in Figure 5.9 enables subgoal refinement to be performed on-line, it is not as general and generic as it could be. To enable completely automatic free action parameter extraction, the planning system should provide the causal links as an output. Unfortunately, causal links are not part of the PDDL output of VHPOP. The complete automation of free parameter extraction has therefore not been implemented yet.

```
Action instantiateAction(abstract_action, params[], context, belief_state) {
  if (abstract_action == "goto") {
 ...
   xg = belief_state.getX(params[1])
   yg = belief_state.getY(params[1])
   if (context == "dribble") {
     phig = [-PI,PI]
     vg = [0.0, 0.3]
   } else if (context == "shoot") {
     phig = [-PI,PI]
     vg = [0.0, 0.6]
   } else {
     phig = [-PI,PI]      // Default value
     vg = [VMIN,VMAX]  // Default value
   }
   return GoToPoseAction(x,y,phi,v,xg,yg,phig,vg)
 } else if ...  (other actions)

instantiateAction("goto", ["start", "ball"], "dribble", belief_state) =>
  goToPose(  x=2.68,  y=2.12, phi=0.21, v=0.4,
            xg=4.17, yg=3.40, phig=[-PI, PI], vg=[0.0, 0.3] )
```

Figure 5.9. An action sequence with free parameters set to ranges that depend on the context.

5.3.2 Optimizing free action parameters

To optimize the action sequence, the system will have to find those values for the free action parameters for which the overall performance of the sequence is the highest. The overall performance is estimated by simply summing over the performance models of all actions that constitute the sequence. We will first demonstrate this process with an example, and then give the general optimization approach.

Examples

In Figure 5.10, Figures 4.8 and 5.1 have been combined. The first two polar plots represent the predicted execution duration of the two individual actions for different values of the angle of approach, a free action parameter. 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 2.1s, 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 7.5s, because the second action takes 5.4s. The third plot clearly shows that this is not the optimum overall performance. The minimum is actually 6.1s, for an angle of 59°. Below the polar plots, the situation of Figure 5.1 is repeated, this time with the predicted performance for each action.

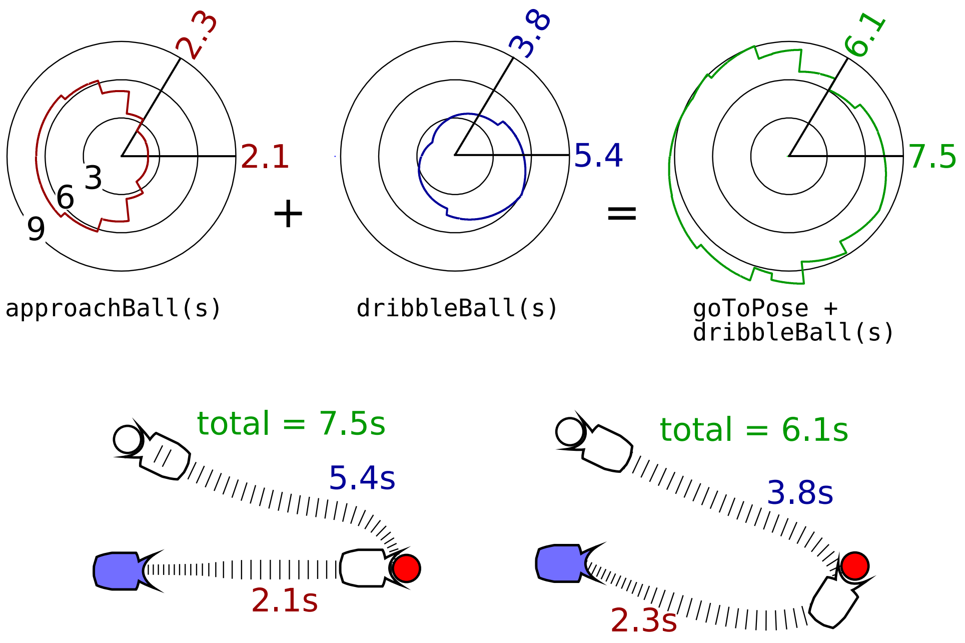


Figure 5.10. Selecting the optimal subgoal by finding the optimum of the summation of all action models in the chain.

Another similar example from the service robotics domain is depicted in Figure 5.11. In principle it is the same scenario as in Figure 5.10, but this time, the target translational velocity has also been added. Of course, the different dynamics of the simulated B21 lead to different execution times for this scenario. The angle of approach qualitatively has the same effect as in the soccer scenario. Note that with higher target translational velocities, the first action can be executed faster, as no braking is required before arriving at the goal. Again, the fastest execution of the first action is at 0°, and the overall fastest execution at 64°, with a maximal target velocity of 0.7m/s.

For reasons of clarity, only one or two parameters were optimized in these examples, and we simply ‘read’ the minima from the plot. Of course, the robots must be able determine this minimum automatically and on-line, possibly with several free action parameters and resulting high-dimensional search spaces. The next sections will present two optimization methods. The first approach is analytical, and only possible with model trees. The second is a genetic algorithm, which is independent of the algorithm with which prediction models have been learned.

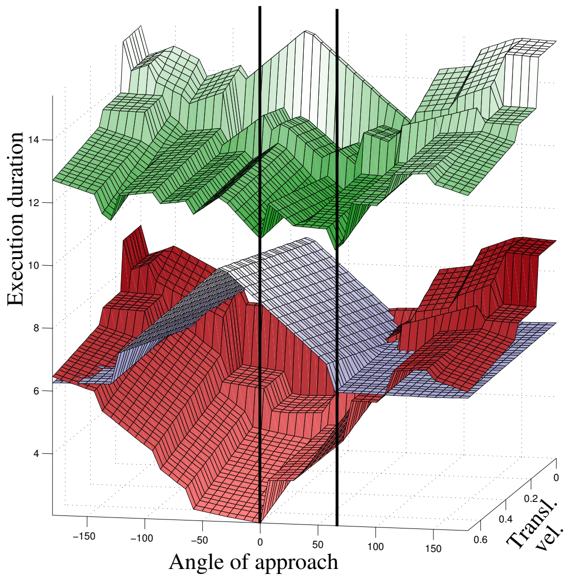


Figure 5.11. Another free action parameter optimization example.

Analytical optimization of Model Trees

In Figure 5.11 the three functions clearly consist of a bounded set of 2-dimensional planes in the 2-dimensional feature space. In general, model trees partition the n-dimensional feature space, and represent the data in each partition with a n-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 m hyperplanes, which leads to m corner minima. The global minimum can then be determined by choosing the minimum of all 'minimal corners'. The computational complexity of this approach is far lower than that of sampling, or other search techniques such as genetic algorithms. The implementation and the complexities are presented in Section C.3 in Appendix C.

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 this one model tree. The implementation is also presented in Section C.3.

Optimization with a Genetic Algorithm

When model tree optimization is not possible, we optimize the free action parameters with a genetic algorithm (GA) (Goldberg, 1989). Our implementation of the GA 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, we first applied it to several optimization benchmarks, such as the De Jong's function, Schwefel's function and Ackley's Path function. The results and optimization times are reported in (Koska, 2006). In the subgoal refinement scenarios to be presented in Section 5.4, the optimization time is usually small in comparison to the gain in performance. For the extreme scenario, where several actions with many free action parameters are optimized, our implementation of the GA still takes less than 0.5s to get a good result.

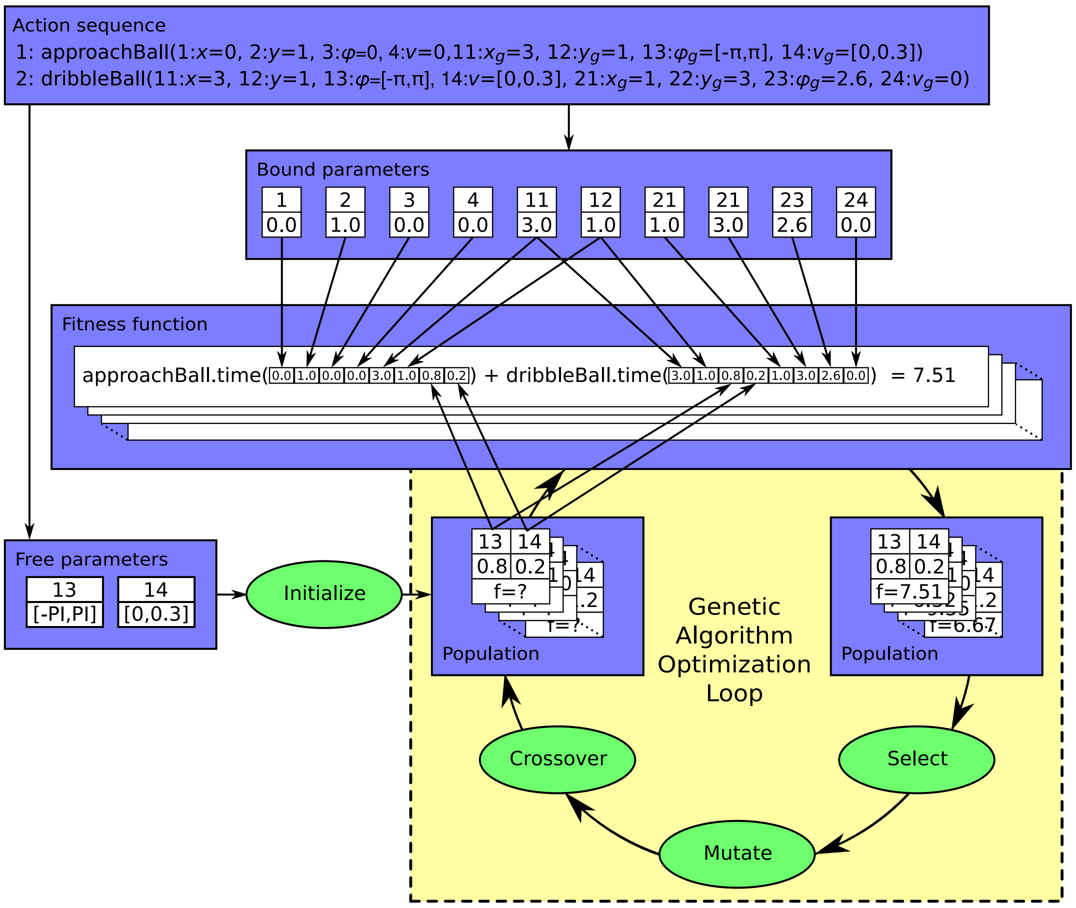

Action sequence
1: approachBall(1:x=0, 2:y=1, 3:φ=0, 4:v=0,11:xg=3, 12:yg=1, 13:φg=[-π,π], 14:vg=[0,0.3])
2: dribbleBall(11:x=3, 12:y=1, 13:φ=[-π,π], 14:v=[0,0.3], 21:xg=1, 22:yg=3, 23:φg=2.6, 24:vg=0)
Bound parameters
1 0.0
2 1.0
3 0.0
4 0.0
11 3.0
12 1.0
21 1.0
21 3.0
23 2.6
24 0.0
Fitness function
approachBall.time(0.0 1.0 0.0 0.0 3.0 1.0 0.8 0.2) + dribbleBall.time(3.0 1.0 0.8 0.2 1.0 3.0 2.6 0.0) = 7.51
Free parameters
13 [-PI,PI]
14 [0,0.3]
Initialize
13 14 0.8 0.2 f=?
Population
Genetic Algorithm Optimization Loop
13 14 0.8 0.2 f=7.51
f=6.67
Population
Select
Mutate
Crossover

Figure 5.12. Optimization in subgoal refinement with a genetic algorithm

Figure 5.12 depicts how the optimization with the GA has been 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). These are used to represent 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’.

The next step is to partition the action parameters in the action sequence into two sets: one set contains action parameters that were bound to certain value during instantiation, and the other set contains the 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 then 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 GA loop is then started. The loop halts if the best fitness has not changed over the last 50 generations, or if 500 generations have been evaluated.

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. Then, 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. This formula has been chosen to guarantee that the fitness is a non-negative number, which is necessary for fitness proportionate selection.

5.4 Empirical Evaluation

In this section, we will introduce the scenarios and action sequences to which subgoal refinement is applied. Then, the results of applying subgoal refinement will be presented.

In the robotic soccer domain, the action sequence to be optimized is the `approachBall` action, followed by a `dribbleBall` action, as in Figure 5.1. 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 were tested. In the first scenario, the goal is to put a cup from one table to the other, which can be achieved by the action sequence depicted in Figure 5.13. In each episode in the evaluation, the topology of the environment in each scenario stays the same, but the initial robot position, the tables and the cups are randomly displaced along the arrows in Figure 5.13. Scenario 2 was a variation of Scenario 1, in which two cups had to be delivered.

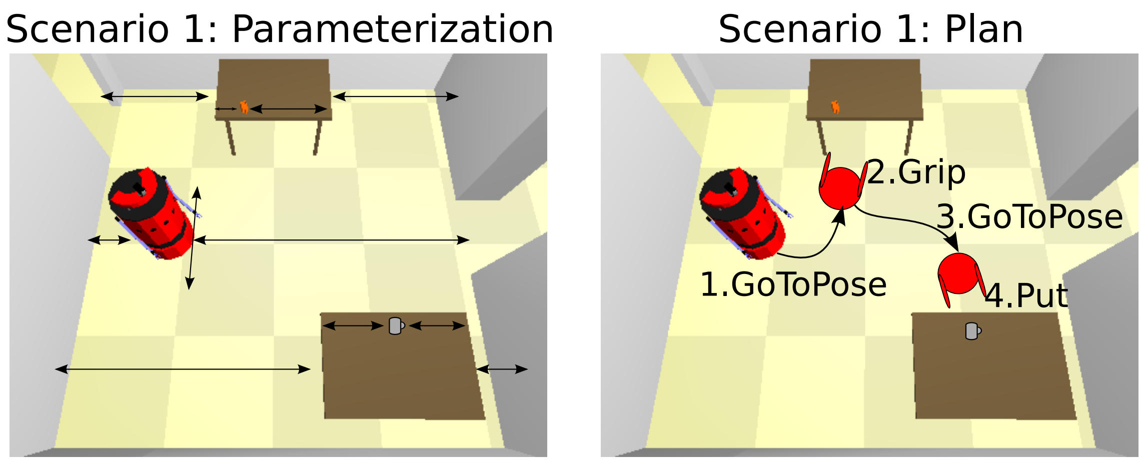
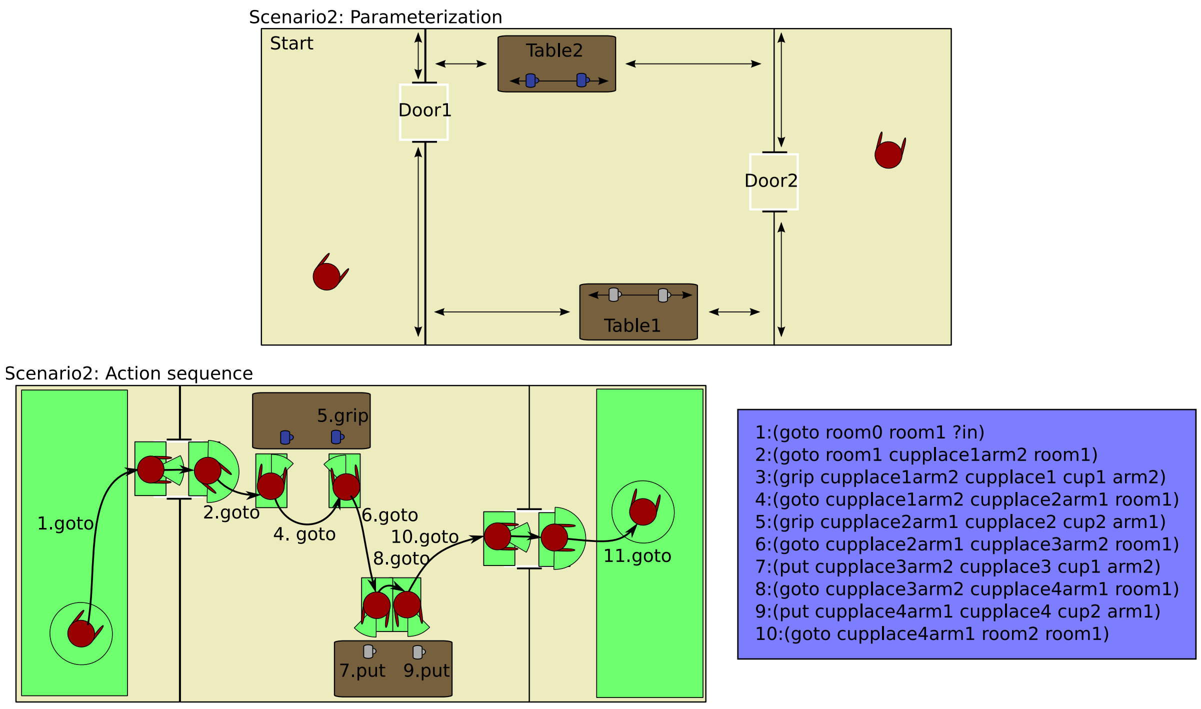


Figure 5.13. Scenario 1. In each episode, the objects and the initial robot position are different. Possible positions are indicated by arrows.

The kitchen scenarios have many free action parameters. Because preconditions usually fix either navigation ór manipulation motions but never both (they are independent), one of these action parameter sets is always free. Furthermore, the distance the robot must have 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 way-points are also not fixed, so free for optimization as well. In Figure 5.14, an example of free action parameters that arise from instantiating a plan in the kitchen scenario are given. The green areas represent these ranges, where square areas represent a range of possible positions, and the circular areas possible angles.

In the arm control domain, sequences of reaching movement were 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 (Stulp et al., 2007), and chose the way-points accordingly. Figure 5.15(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 were used, as depicted in Figure 5.15(b). The free action parameters are the angular velocities at these way-points.



```
1:(goto room0 room1 ?in)
2:(goto room1 cupplace1arm2 room1)
3:(grip cupplace1arm2 cupplace1 cup1 arm2)
4:(goto cupplace1arm2 cupplace2arm1 room1)
5:(grip cupplace2arm1 cupplace2 cup2 arm1)
6:(goto cupplace2arm1 cupplace3arm2 room1)
7:(put cupplace3arm2 cupplace3 cup1 arm2)
8:(goto cupplace3arm2 cupplace4arm1 room1)
9:(put cupplace4arm1 cupplace4 cup2 arm1)
10:(goto cupplace4arm1 room2 room1)
```

Figure 5.14. Examples of free action parameter ranges in a kitchen scenario

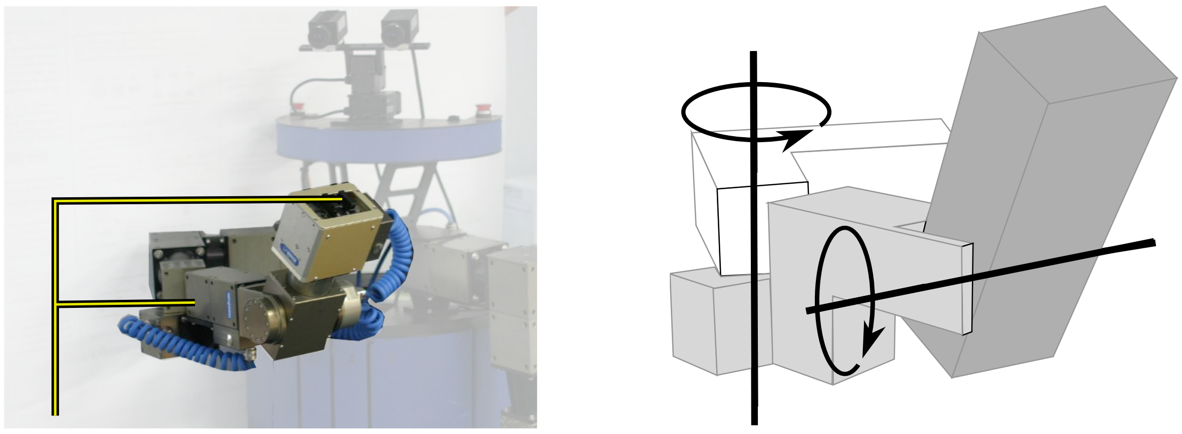
5.4.1 Results

Table 5.1 lists the results of applying subgoal refinement to the different domains and scenarios, where a is the number of action in the sequence, and e is the number of episodes tested.

Scenario	a	e	$\overline{t_{h=1}}$	$\overline{t_{h=2}}$	$\overline{t_{h=2}/t_{h=1}}$	p
Soccer (Simu.)	2	1000	9.8s	9.1s	6.6%	0.00
Soccer (Real)	2	100	10.6s	9.9s	6.1%	0.00
Kitchen (Sc. 1)	4	100	46.5s	41.5s	10.0%	0.00
Kitchen (Sc. 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 5.1. Subgoal refinement results

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 have chosen 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. The execution time of a single



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

(b) The two degrees of freedom used for drawing.

Figure 5.15. Arm control domain experiment.

action is denoted t, which has three indexes referring to the horizon, the episode, and the action in the sequence. For instance $t_{1,64,2}$ refers to the second action in the 64th episode, that was performed with a horizon of 1, which is greedy. The mean overall execution duration over all episodes is denoted $\overline{t_{h=1}}$, and computed using Equation 5.1.

Since subgoal refinement optimizes the execution duration of the current ánd next action, it has a horizon of 2. The fourth columns lists the mean overall execution duration with subgoal refinement $\overline{t_{h=2}}$, which is computed with an equation equivalent to Equation 5.1 with $h = 2$.

The improvement achieved with subgoal refinement in episode e is computed using Equation 5.2, and the mean over all episodes is computed using Equation 5.3[2].

$$\overline{t_{h=1}} = \frac{1}{n}\sum_{p=1}^{n}\sum_{a=1}^{m} t_{1,p,a} \tag{5.1}$$

$$t_{h=2,p=j}/t_{h=1,p=j} = (1 - \frac{\sum_{a=1}^{m} t_{2,j,a}}{\sum_{a=1}^{m} t_{1,j,a}}) \tag{5.2}$$

$$\overline{t_{h=2}/t_{h=1}} = \frac{1}{n}\sum_{p=1}^{n}(1 - \frac{\sum_{a=1}^{m} t_{2,p,a}}{\sum_{a=1}^{m} t_{1,p,a}}) \tag{5.3}$$

The fifth column in Table 5.1 lists the mean improvement achieved with subgoal refinement $\overline{t_{h=2}/t_{h=1}}$. The p-value of the improvement was computed using a dependent *t*-test with re-

[2]In (Stulp and Beetz, 2005b), improvements were computed with $1 - \overline{t_{h=1}/t_{h=2}}$.

peated measures, as each episode has been tested twice, once with, and once without subgoal refinement. A significant and substantial 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 5.16. The angular velocities were set to zero (upper row) or optimized with subgoal refinement (lower row). The axes represent the angles of the two joints. This figure demonstrates well that the trajectories ares smoother with subgoal refinement: the arms draws one long stroke, rather than discernible line segments. Since the arm control domain was mainly included for visualization purposes, there are only a few episodes. For this reason the overall improvement is not significant (>0.05).

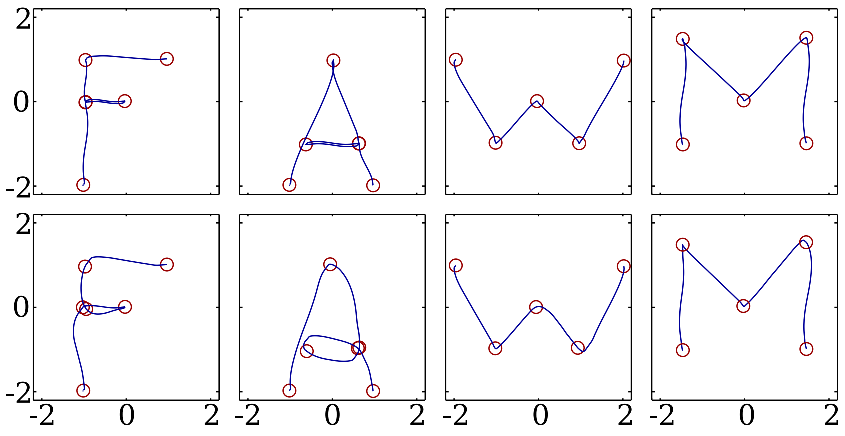


Figure 5.16. Drawing letters without (upper row) and with (lower row) subgoal refinement. With refinement, letters are drawn faster and smoother.

Although optimizing speed also leads to smoother motion in this domain, for humans it has been shown that variability minimization is a more likely cause for smooth arm motion (Simmons and Demiris, 2004). In this chapter, the main goal is not to explain or model human motion, but rather to demonstrate the effects of optimizing sequences of actions. Interestingly enough, Simmons and Demiris (2004) have also used their methods to draw letters with smooth writing motions (Dearden, 2006).

5.4.2 Influence on individual actions

Table 5.2(a) and Table 5.2(b) demonstrate the effect of subgoal refinement on individual actions in the action sequence. The mean execution duration of each action over all episodes was computed using Equation 5.4.

$$\overline{t_{h=2,a=k}} = \frac{1}{n}\sum_{p=1}^{n} t_{2,p,k} \tag{5.4}$$

The table to the left lists the execution of the individual action of Scenario 1 from the service robotics domain[3]. The right table lists the same from a scenario from the soccer domain. In this scenario, the simulated soccer robot navigates to four way-points on the field with the `goToPose` action, as depicted in Figure 5.17. At each way-point the angle of approach and translational velocity are optimized. This scenario was also executed in 100 episodes with different randomly placed way-points in each episode.

Action		$h=1$	$h=2$
$a=1$	(`gotoPose`)	4.4s	5.7s
$a=2$	(`grip`)	20.8s	18.5s
$a=3$	(`gotoPose`)	5.9s	5.1s
$a=4$	(`put`)	15.4s	12.2s
$a=1..4$	(total)	46.5	41.5

(a) Service robotics domain.

Action		$h=1$	$h=2$
$a=1$	(`gotoPose`)	4.2s	4.8s
$a=2$	(`gotoPose`)	6.0s	4.9s
$a=3$	(`gotoPose`)	5.8s	5.6s
$a=4$	(`gotoPose`)	6.7s	5.0s
$a=1..4$	(total)	22.7s	20.3s

(b) Soccer domain.

Table 5.2. Influence of subgoal refinement on the execution duration of individual actions in a sequence.

A clear effect on the individual actions is that the execution duration of the first action is slower with subgoal refinement, allowing the faster execution of the other actions. The difference is most striking in the last action in the table on the left. In the greedy approach, the trouble the robot has caused itself by optimizing three actions greedily often culminates in a very awkward position to execute the last action.

5.4.3 Sequences with more actions

In Figure 5.17, an example episode from the soccer scenario from Section 5.4.2 is depicted. Here the robot has to traverse four way-points with the `goToPose` action. 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

[3]The grip and put actions take more time than in Table 4.2, because the actual closing and opening of the gripper at the end of each reach action has been incorporated into the action. This additional time was constant, and not taken into consideration during optimization.

the next $h > 2$ actions, as indicated by the colors in Figure 5.17. The higher the horizon h the more subgoal refinement is preparing for actions further in the future.

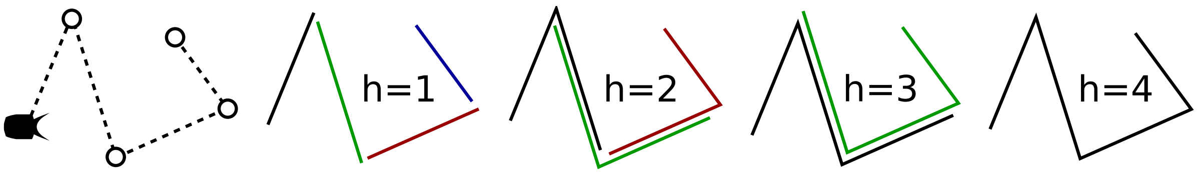


Figure 5.17. Visualization of the horizon h in subgoal refinement.

To evaluate the effect of optimizing more than two actions quantitively, sequences of four actions were optimized using subgoal refinement with different horizons. The two scenarios from Section 5.4.2 were used: the soccer scenario depicted in Figure 5.17 and the kitchen scenario depicted in Figure 5.13. The results are summarized in Table 5.3. 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 5.3. Effect of the subgoal refinement horizon h on performance improvement.

Intuitively, the effect of future actions on the current action should decrease, the further the future action lies in the future. For instance, your position at the table will influence the time it will take to grab the cup on this table, as well as the time it will take to navigate to the next room. However, it will not likely influence the time needed to put down the cup in the next room. It is interesting to see that the substantial improvement in both scenarios indeed diminishes quickly after $h = 2$. Whereas a significant but only marginal improvement is sometimes still to be had from $h = 2$ to $h = 3$, the improvement to $h = 4$ is not significant anymore.

5.4.4 Predicting performance decrease

There are many cases in which subgoal refinement does not have an effect. For instance, if the robot, the ball and the final destination are perfectly aligned, there is not much to be had from subgoal refinement, as the greedy approach already delivers the optimal angle of approach: straight toward 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 better than subgoal refinement. To evaluate these effects, 1000 episodes where executed in simulation with both $h = 1$ and $h = 2$. Then, the overall improvement (6.6%) was separated into episodes in which subgoal refinement improved (`+`), kept equal (`0`), or made worse (`-`) the execution duration, as listed in Table 5.4

Before filtering	Total	+	0	-
#episode	1000	573	267	160
improv	6.6%	16.2%	0.0%	-17.1%

Table 5.4. Positive and negative influence of subgoal refinement on execution duration.

This result shows that the performance improved in 573 cases, and in these cases causes a 16.2% improvement. In 267 cases, there was no improvement. This is to be expected, as there are many situations in which the three positions are already optimally aligned (e.g. in a straight line), and subgoal refinement will have no effect. Unfortunately, applying our method also causes a decrease of performance in 160 out of 1000 episodes.

To analyze in which cases subgoal refinement decreases performance, we labeled each of the above episodes `+`, `0` or `-`. 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, as can be seen in the confusion matrix of the learned decision tree in Table 5.5. 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 will see another example of such a *meta action model* in Section 7.4.2.

The rules and a graphical representation are depicted in Figure 5.18. In this graph, the robot always approaches the centered ball from the left at different distances. The different regions indicate whether the performance increases, decreases, or stays equal. Three instances with different classification and therefore different colors circles have been inserted. The trajectories are a qualitative indication of the robot motion.

The rules declare that performance will stay equal if the three points are more or less

		Predicted				
		+	0	-		Totals
	+	48.6%	1.4%	1.5%	→	51.5%
Actual	0	8.1%	28.0%	0.8%	→	36.9%
	-	1.4%	0.2%	10.2%	→	11.8%
		↓	↓	↓	↘	
Totals		58.1%	29.6%	12.5%		86.7%

Table 5.5. Confusion matrix of the decision tree that predict performance decrease

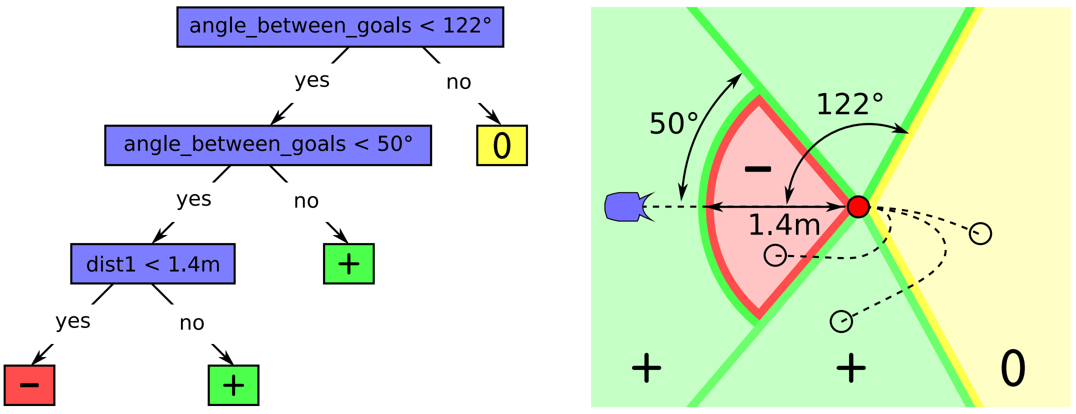


Figure 5.18. The decision tree that predicts whether subgoal refinement will make the performance better, worse or have no influence at all.

aligned, and will only decrease if the final goal position is in the same area as which the robot is, but only if the robot's distance to the intermediate goal is smaller than 1.4m. Essentially, this last rule states that the robot using the `goToPose` action has difficulty approaching the ball at awkward angles if it is close to it. In these cases, small variations in the initial position lead to large variations in execution time, and learning an accurate, general model of the action fails. The resulting inaccuracy in temporal prediction causes suboptimal optimization. Note that this is a shortcoming of the action itself, not of subgoal refinement. The meta action model of applying subgoal refinement is essentially telling us that subgoal refinement is working fine, but that the `approachBall` is rather non-deterministic under certain conditions, and needs improvement.

We then performed another 1000 test episodes, as described above, but only applied subgoal refinement if the decision tree predicted applying it would yield a higher performance. The results are summarized in Table 5.6. The performance improvement due to subgoal refinement was 6.6%, and is now 8.6% (p-value is 0.000). More importantly, the number of cases in which performance is worsened by applying subgoal refinement has decreased from 160 (16.0%) to 54 (5.4%). Apparently, the decision tree correctly filters out cases in which applying subgoal refinement would decrease performance. Note that when performance is decreased, it is not so dramatic anymore (-17.1%⇒-10.1%): the decision tree is filtering out the worst cases.

After filtering	Total	+	0	-
#episode	1000	557	389	54
improv	8.6%	16.4%	0.0%	-10.1%

Table 5.6. Positive and negative influence of subgoal refinement on execution duration, *after* filtering for cases where a decreased performance is predicted.

5.5 Related Work

5.5.1 Smooth motion as an explicit goal

Many behavior based approaches also achieve smooth motion by a weighted mixing of the motor commands of several actions (Jaeger and Christaller, 1998; Saffiotti et al., 1993). In these approaches, there are no discrete transitions between actions, so they are also not visible in the execution. In computer graphics, the analogous approach is called *motion blending*, and is also a wide-spread method to generate natural and fluent transitions between actions, which is essential for lifelike animation of characters. Perlin (1995) presents visually impressive results. More recent results are described by Shapiro et al. (2003) and Kovar and Gleicher (2003). Since there are no discrete transitions between actions, they are also not visible in the execution. In all these blending approaches, achieving optimal behavior is not an explicit goal; it is left to chance, not objective performance measures.

Hoffmann and Düffert (2004) have developed a very different technique for generating smooth transitions between skills for the AIBO quadruped robots. The periodic nature of robot gaits allows their meaningful representation in the frequency domain. Interpolating in this domain yields smooth transitions between walking skills. Since the actions we use are not periodic, these methods do unfortunately not apply.

5.5.2 Classical planning

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 would have an easier time representing time constraints and resources, most could not deal with the action choices in this problem. Systems that integrate planning and scheduling, such as (Smith et al., 2000), are able to optimize resources, but ignore interactions between actions and intermediate dynamical states, so do not apply well to continuous domain problems.

Least commitment planning also depends on the concept of unbound variables (Weld, 1994). The idea is to keep variables unbound as long as possible, and bind them only when is necessary. This makes plans more flexible, and plan execution more robust. However, variables that are never bound, are still unbound in the final plan. It exactly these that we use for optimization.

Refinement planning is a method whose name bears similarities with subgoal refinement, but which describes another process (Kambhampati et al., 1995). Refinement planning searches for an action sequence that will achieve the goal by pruning away action actions sequences that will not. Initially, all action sequences are considered solutions. Subsequent refinement operations then narrow the set of possible action sequences by adding constraints to it. Our system does not refine the plans themselves to find action sequences, but rather the execution of the plans, given a certain action sequence. Although resources are sometimes represented during planning, planning in general is only interested in finding a plan that is *valid*. Our system takes a valid plan, and finds a plan execution that is *optimal*, with respect to the predicted performance. In principle, a refinement planning system could be used in the "planning system" module in Figure 5.4.

In PDDL (Fox and 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 have been 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. We are not aware of other planning systems that generate abstract plans and simultaneously optimize the actual physical behavior of robots.

5.5.3 Symbolic planning with action execution

Bouguerra and Karlsson (2005) describe a computational model that is quite similar to ours. In their model, the abstract plan domain is called "Deliberation", and the action execution and sensing process is provided by the "ThinkingCap" robot-control architecture. The interface between the two is called the "Anchoring" modules. There are two important differences between their models and ours. First of all, probabilistic planners are used in the abstract planning domain, being BURIDAN (Kushmerick et al., 1994) and PTLPlan (Karlsson and Schiavinotto, 2002). Therefore, their system can deal with uncertain worlds. The other enhancement is plan failure recognition and plan repair. Because the focus of this dissertation is acquiring and applying action models to tailor actions to task contexts, we deliberately abstract away from these enhancements. Note that our methods are in no way incompatible to the ones described in (Bouguerra and Karlsson, 2005), and merging both approaches would combine the benefits of both, as discussed in Section 5.2.3.

aSyMov (Cambon et al., 2004) is another approach that bridges the gap between symbolic planning and plan execution, in complex simulated 3-D environments. The main goal is to reason about geometric preconditions and consequences of actions. This is done by defining a Configuration Space, in which constraints on mobile robots and objects can be expressed. Then, symbols representing locations in the world are related to constraints in Configuration Space. This allows the specification of not only at a symbolic level but also with regard to the geometry of an environment. The input of the planner is 1) a symbolic data file, specified in PDDL 2) the geometric data 3) and a semantic file that relates symbols to geometric data. Symbolic planning is done with the Metric-FF (Hoffmann, 2003) system, and geometric planning is done with the Move3D library (Siméon et al., 2001). The aSyMov library merges the result of both using the semantic file.

Here again, we see great potential for merging aSyMov and subgoal refinement, as they are complementary, rather mutually exclusive, as discussed in Section 5.2.3. Cambon et al. (2004) actually mentions that the resulting plan is improved and optimized in some way, but does not describe how. Subgoal refinements might very well be integrated in this optimization step.

Hierarchical Reinforcement Learning (Barto and Mahadevan, 2003), which was introduced in Section 3.1.3 also optimizes actions and action sequences, by maximizing the expected reward. In most of these approaches, the action sequences or action hierarchies are fixed (Parr, 1998; Andre and Russell, 2001; Dietterich, 2000; Sutton et al., 1999). The only approach we know of that explicitly combines planning and Reinforcement Learning is RL-TOPS (*Reinforcement Learning - Teleo Operators*) (Ryan and Pendrith, 1998). In this approach, sequences of actions are first generated based on their pre- and postconditions, 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. RL-TOPS has been tested in grid worlds and also more complex domains (Ryan and Reid, 2000), but not in the context of mobile robotics. A more recent RL-planning hybrid is presented in (Grounds and Kudenko, 2005), though it is not clear how this work extends the work of Ryan et al. In general, the benefit of action models over Reinforcement Learning were discussed in Section 4.4.1.

Model trees have learned as performance models, and used to optimize Hierarchical Transition Network (HTN) plans (Belker et al., 2003; Belker, 2004). This work was already introduced in Section 3.2.3. HTN plans are structured hierarchically from high level goals to the most low level commands. To optimize performance, the order of the 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

XFRMLearn is an approach that also elegantly combines declarative and learned knowledge to improve the performance of robot navigation execution (Beetz and Belker, 2000). The XFRMLearn system optimizes plans through plan transformation, which is closely related to subgoal assertion, which will be presented in the next chapter. Therefore, we postpone the discussion of this work to Section 6.5.1.

5.5.4 Motion planning and execution

Generating collision-free paths from an initial to a final robot configuration is also known as robot motion generation. A common distinction between algorithms that generate such paths is:

Global approaches These approaches determine a path to the goal off-line before execution, based on a global snap-shot of the world. Because a global view of the world is known, global constraints such as obstacles can be taken into account, and ending up in a local minimum can be avoided. The problem such global algorithms solve is called the basic motion planning problem. On-line, the predetermined path is the executed to actually achieve the goal configuration. Therefore, the environment may not change during execution, as this could invalidate the predetermined path. Examples of this approach are roadmaps and cell decomposition.

Local approaches To adapt to local changes, local approaches use sensory information to direct there motion on-line during execution. This enables the avoidance of obstacles. Due to their local perspective, such approaches can get stuck into local minima, such as a dead-end in a corridor. An example of this method is the potential fields approach.

Hybrid approaches By combining both local and global approaches, hybrid methods hope to get the best of both worlds. Examples are the Elastic Strips framework (Brock and

Khatib, 1999), and the planning system and execution system of GOFER (Choi and Latombe, 1991). Both will be discussed more elaborately in the next section.

Brock and Khatib (1999) give an overview and examples of all three approaches. The methods presented in this chapter are a global approach, as the planning is performed off-line, before plan execution. This means that failures in action execution, for instance due to unforeseen or dynamic obstacles require replanning. The same holds for all global approaches, of which (Bouguerra and Karlsson, 2005; Cambon et al., 2004) are examples discussed in the previous section. Whereas motion planning and execution algorithms focus on collision-free paths in configuration space, our methods deal with the more general problem of mapping symbolic plans to executable action sequence. Declarative plans allow for higher levels of abstraction than standard motion planning techniques, which facilitates the design of abstract actions and common-sense constraints.

In the Elastic Strip Framework (ESF) (Brock and Khatib, 1999) approach, a set of spheres, determined heuristically, defines the local free space around a configuration of a robot. Along a trajectory determined with a global motion planning algorithm, a sequence of configurations is chosen, which together are called the elastic strip. The unification of the local free spaces around these configurations is called the *elastic tunnel*. Obstacles exert external repulsive forces represented by potential fields on the elastic strip, causing it to stretch. As this stretching does not affect the topology of the strip, the global constraints of the motion plan are satisfied, and local minima are avoided. Choi and Latombe (1991) describe the planning and execution system of the mobile robot GOFER, which could be considered a predecessor of the Elastic Strip framework. Here, *channels* are rectangular areas which the robot should traverse in order to reach the goal. Within these rectangles, the robot is free to move, for instance to avoid obstacles.

The quote *"The elastic tunnel can be imagined as a tunnel of free space within which the trajectory can be modified without colliding with obstacles."* shows the conceptual similarity of the elastic tunnel with free action parameters. The main difference is that ESF exploits freedom in the parameterization of the path to the goal, to adapt on-line to unforseen events during action execution, whereas we exploit freedom in the parameterization of the goal itself, to optimize the expected performance off-line. The path is not considered explicitely, but rather emerges on-line when the action is executed.

5.5.5 Redundancy resolution

In human motor control, there is a distinction between the external space, which can be expressed in terms of task coordinates, and the internal space, which refers to the internal coordinates of the muscle system. In most motor tasks, the number of degrees of freedom in the internal space far exceeds that in the external space. The internal space therefore has a high level of redundancy with respect to the external space. Put simply: there are many ways you can bring a glass of water to your lips of which, in the words of Wolpert and Ghahramani (2000), some are sensible and some are silly.

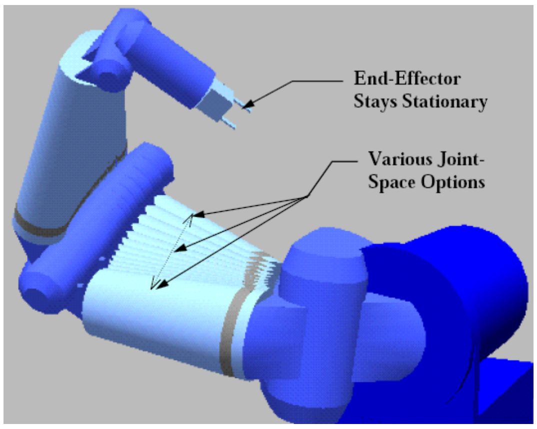


Figure 5.19. In robotic arm control, actions are often redundant. Picture taken from (Hooper, 1994).

The reason why we typically witness stereotypical 'sensible' movement is because the redundancy can be exploited to optimize 'subordinate criteria' (Schaal and Schweighofer, 2005), or 'cost functions' (Wolpert and Ghahramani, 2000), such as energy efficiency or variance minimization. This process is called redundancy resolution. In cognitive science, the goal is often to determine the cost function that is being optimized, given the empirical motion data (Wolpert and Ghahramani, 2000).

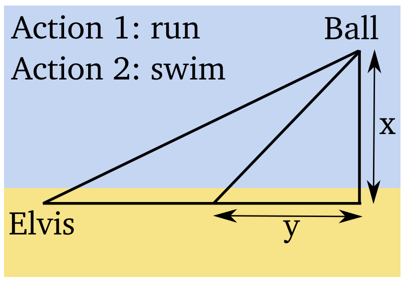
Redundancy resolution has been well studied in the context of robot arm control. Arm poses are said to be redundant if there are many arm configurations that can achieve the same task, as depicted in Figure 5.19. All these configurations are called motion or null space, and finding the best configuration is called null-space optimization, which is equivalent to redundancy resolution. Hooper (1994) proposes to use direct search methods to find the configuration with the best fault tolerance in motion space. This approach is analytical, and the implementation specific to arm domains. Since we learn our models from observed experience, they can basically be applied to any robotic task, from mobile robot navigation to arm control. Nakanishi et al. (2005) gives an overview and experimental evaluation of various other null-space optimization techniques.

5.5.6 Subgoal Refinement

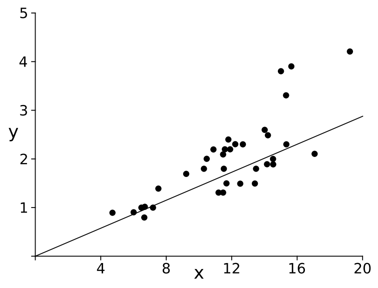
Most similar to our work, from the point of view of smoothness as an emergent property of optimality requirements, is the approach of Kollar and Roy (2006). Here a simulated robot maps its environment with range measurements by traversing a set of way-points. Reinforcement learns a policy that minimizes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

On a more light-hearted note, we are happy to report that some evidence (if interpreted correctly) shows that dogs are capable of performing subgoal refinement. Figure 5.20(a) depict Elvis the Dog. Elvis is owned by Hope College (Michigan, USA) professor Tim Pennings. Prof. Pennings often takes Elvis to the beach to fetch tennis balls from the water, as in Figure 5.20(a). Elvis achieves this by first running along the beach (action 1), and the swimming to the ball (action 2). Because running is much faster than swimming, the optimal policy is not to go to the ball in a straight line, but rather run parallel to the beach for a certain distance, and then swim to the ball, as in Figure 5.20(b). Which distance this should be is a standard optimization problem, often found in college tests. Interestingly enough, Elvis seems to be solving this problem as he chooses the mathematical optimal distance in varying scenarios. By measuring Elvis' running and swimming speed, Prof. Pennings could plot the optimal distance as in Figure 5.20(c), taken from (Pennings, 2003). The distance that Elvis actually chooses (the dots in the graph represent individual fetch trials) matches this optimal line quite closely.

(a) Elvis the dog.



(b) The optimization problem.



(c) Empirical data.

Figure 5.20. Elvis the dog solves the ‘beach optimization problem’.

Why is this subgoal refinement? Because Elvis is choosing the intermediate goal (i.e. the point where he enters the water) such that the overall execution of the action sequence is optimized. The scenario is very similar to the example in Figure 5.1, where performing the first action suboptimally leads to a better overall performance.

(Gallego and Perruchet, 2006) have challenged the notion that Elvis is performing global optimization, and give a much simpler local optimization strategy that solves the optimization problem with the same result: go into the water as soon as the relative speed of the ball to my own position is lower than my swimming speed. However, further experiments have shown that this can also not be the whole story (Pennings, 2007), and the debate continues.

5.6 Conclusion

Durative actions provide an conceptual abstraction that is reasoned about either by the designer during action selection design, or, if the abstraction has been explicitly coded into the controller, by the action selection system itself. Action abstractions partially achieve their abstraction by not taking into account all action parameters. Although these free action parameters are not relevant to the action on an abstract level, they often are relevant to the performance of executing the plan. Subgoal refinement was implemented using the Planning Domain Description Language, and the standard partial order causal link planner VHPOP. We have shown how these free action parameters can be extracted, and optimized analytically or with genetic algorithms, with respect to expected performance computed by action models.

Without subgoal refinement, the transitions between actions were very abrupt. In general, these motion patterns are so characteristic for robots that people trying to imitate robotic behavior will do so by making abrupt movements between actions. It is interesting to see that requiring optimal performance can implicitly yield smooth transitions in robotic and natural domains, even though smoothness in itself is not an explicit goal in either domain.

We believe this is an important contribution towards bridging the gap between robot action execution on the one hand, and planning systems and deliberative components in general on the other. Subgoal refinement combines abstract human-specified knowledge with learned predictive knowledge. As robots are becoming more dextrous, and their actions more expressive, abstraction will become more important for keeping action selection and planning tractable. This also means the gap between an action's abstraction and its execution will widen, and more free action parameters will arise. Suboptimal performance and jagged motion is an unavoidable consequence of leaving these free action parameters unconsidered. Subgoal refinement not only contemplates free action parameters, but exploits them by optimizing them with respect to the expected overall performance, thereby turning the curse of free action parameters into a blessing.

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 both 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 constant operation time 15 minute, speed is far more important then energy consumption. In service robotics it is the other way around.

The results reported in this chapter have been published in: (Stulp et al., 2007; Stulp and Beetz, 2006; Stulp et al., 2006b; Stulp and Beetz, 2005b,c,a). Summaries of these publications are given in Appendix D.

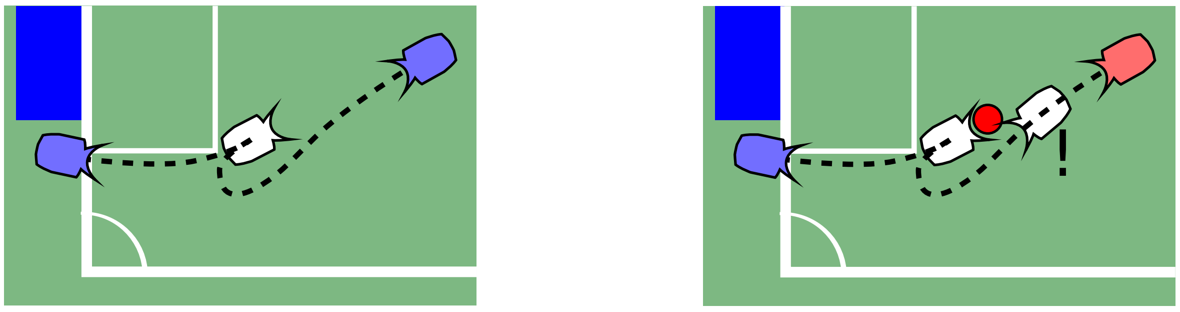
6. Task Context: Task Variants

“Before we learn how to run, we must first learn how to walk.”
English proverb

In order to adapt to new environments and acquire new skills autonomously, robots must be able to learn. Learning generates new knowledge from experience through experimentation, observation and generalization. In practice, learning hardly starts from scratch, and knowledge about previously learned skills can be transfered to novel skills, as Vilalta and Drissi (2002) describe: “Learning is not an isolated task that starts from scratch every time a new problem domain appears.”. Thrun and Mitchell (1993) call this *life-long learning*. Principle IV from Section 1.1 also adheres to this view, as it poses that existing action can be tailored to novel task contexts.

Let us again take an example from soccer. For both humans and robots, approaching a ball is very similar to navigating without considering the ball. Both involve going from some pose to another pose on the field as in Figure 6.1, and both should be implemented to execute as efficiently and fast as possible. However, there are also slight differences between the objective functions for these two tasks. When approaching the ball it is important to not bump into it before achieving the desired pose, as depicted in Figure 6.1(b).

This scenario can be described well in terms of the actions `approachBall` and `goToPose`. The required action for this task is `approachBall`, which is very similar to the `goToPose` action. However, since `goToPose` is not aware of the ball, it often collides with the ball before achieving the desired pose. In fact, we shall see that this action causes a collision more than half the time. To solve this problem, one could write a new action, e.g. `approachBall`. It would probably be very similar to `goToPose`, but take the ball into account. Instead of designing `approachBall` from scratch, it would be better if the robot reused the similar `goToPose`, and adapt it to the current context. For instance, although `goToPose` fails at ball approach more than half the time, it also *succeeds* at doing so quite often. In these cases, it could be reused without change.



(a) Both robots achieve the desired state with `goToPose`.

(b) When approaching the ball, one of the robots bumps into the ball before achieving the desired state with `goToPose`.

Figure 6.1. Similarities and differences between standard navigation and ball approach.

The key to reuse is therefore being able to predict when the action will fail, and when it will succeed. 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. This intermediate state between the actions is a new subgoal. This approach is therefore called subgoal assertion.

In Figure 6.2, the action variant context has been highlighted within the system overview.

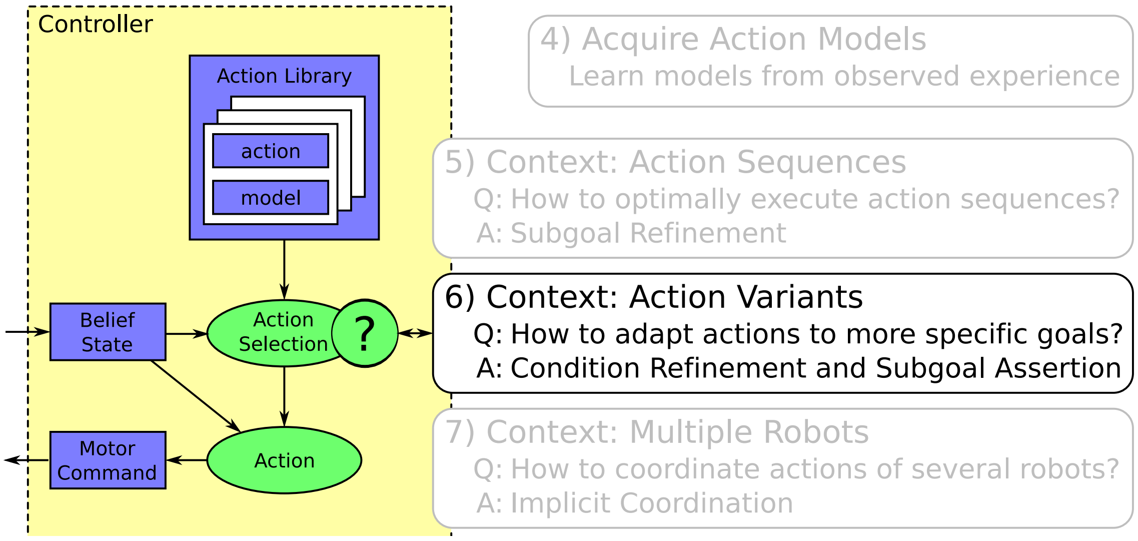

Controller
Action Library
action
model
Belief State
Action Selection
?
Motor Command
Action
4) Acquire Action Models
Learn models from observed experience
5) Context: Action Sequences
Q: How to optimally execute action sequences?
A: Subgoal Refinement
6) Context: Action Variants
Q: How to adapt actions to more specific goals?
A: Condition Refinement and Subgoal Assertion
7) Context: Multiple Robots
Q: How to coordinate actions of several robots?
A: Implicit Coordination

Figure 6.2. Condition refinement and subgoal assertion within the system overview.

In the next section, we will introduce the computational model of subgoal assertion. Then, we will demonstrate how the required failure prediction model can be learned in Section 6.2.

The actual process subgoal assertion will be presented in in Section 6.3. As we shall see, there is an interesting relation between subgoal assertion and subgoal refinement. An empirical evaluation of subgoal assertion is provided in Section 6.4.

6.1 Computational Model

Adapting actions to to refined goals takes place in a two-step procedure: condition refinement and subgoal assertion. These will now be explained.

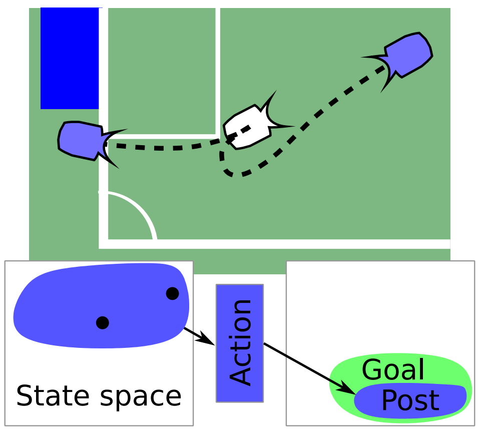
Figure 6.3(a) depicts two possible initial states of a robot in blue, and a goal state in white. In both cases, a single `goToPose` action suffices to bring the robot from the initial state to the goal state. Below the scenario, the general case has been depicted as a transition from the pre-condition to the post-condition of an action. Since the post-conditions satisfies the goal, the action can achieve the goal. The two point in the pre-condition represent two unique states in the state space, for instance those depicted in the field.

Figure 6.3(b) is basically a repetition of the same scenario, but this time the goal is that the robot is at the same position, in possession of the ball. In general, this means that the new goal of approaching the ball is a subset of the former goal of simply navigating there. When executing `goToPose`, the robot to the left achieves at approaching the ball, but the robot to the right does not, as it bumps into the ball beforehand. In general, this is the case because the post-condition of `goToPose` no longer satisfies the refined goal, as can be seen below the field.

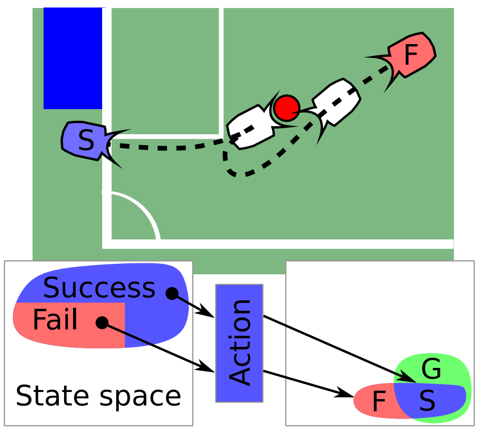
The post-condition of `goToPose` can now be partitioned into a subset which does satisfy the new refined goal, and a subset which does not. These have been represented with blue and red respectively. Analogously, the pre-conditions can be partitioned into a subset `Success` which leads to a final state which is in the subset of the post-conditions that satisfy the refined goal, and a subset `Fail` for which this is not the case. Because the post-, and consequently, pre-conditions of an action are refined for a new task, this is known as *condition refinement*.

As we shall see, the refined pre-condition (the `Success` subset) can be learned. Once the refined pre-condition 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 can be executed as is. For instance, the robot to the left can simply execute the `goToPose` action, as it is in the refined pre-condition.

The robot to the right however is not. 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 can be used again, to take the robot from



(a) Both initial states satisfy satisfy the pre-condition, so executing `goToPose` will lead to successful completion in both cases.



(b) Since the goal has changed, not all states in the post-condition satisfy the goal. Therefore, executing `goToPose` does not lead to the goal for all states in the pre-condition.

Figure 6.3. Computational model of condition refinement.

the `Fail` subset to the `Success` subset. Once this is done, a `goToPose` action that *will* succeed at approaching the ball can be executed.

Summarizing: if an action is predicted to succeed in a novel context, execute it as is. if it is predicted to fail, assert a subgoal from which the action *will* succeed, and execute an extra action to achieve this subgoal.

One issue remains open. In the running example there are infinitely many subgoals from which approaching the ball will succeed. Any state from the `Success` subset could be chosen, but which one is the best? Fortunately, this problem was already posed and solved in Chapter 5. Choosing the best subgoal from many is done using subgoal refinement, as will be explained in Section 6.3.

6.2 Condition Refinement

The key to subgoal assertion is being able to determine if an initial state is in the `Success` or `Fail` subset, which essentially means being able to predict if an action will succeed or not. Determining these subsets manually is a difficult task, as the dynamics and shape of the robot as well specific characteristics of the action come into play. The complex interaction of these features over an extended time makes manual specification near impossible.

Therefore, the robots again learn an action model from experience. To acquire experience,

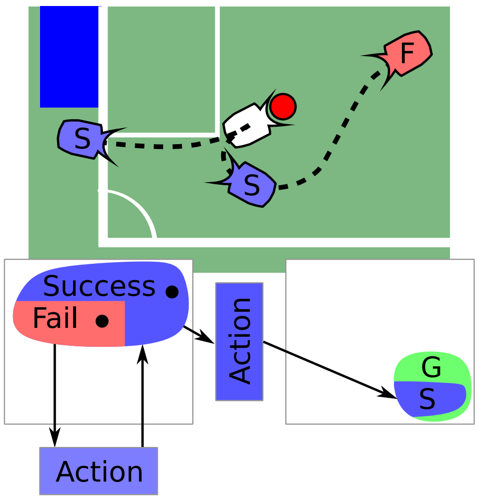


Figure 6.4. Computational model of subgoal assertion

the robot executed `goToPose` a thousand times, with random initial and goal poses. The ball is always positioned at the destination pose. The initial and goal pose were stored, along with a flag that was set to `Fail` if the robot collided with the ball before reaching its desired position and orientation, and to `Success` otherwise. The feature space was the same as for learning the temporal prediction model of `goToPose`, as listed in Table 4.1.

The learned tree, as well as a graphical representation of it, are depicted in Figure 6.5. The goal pose is represented by the robot, and different areas indicate if the robot can reach this position with `goToPose` without bumping into the ball first. Remember that `goToPose` has no awareness of the ball at all. The model simply predicts when its execution leads to a collision or not. Intuitively, the rules seem correct. When coming from the right, for instance, it can be seen that the robot always clumsily stumbles into the ball, long before reaching the desired orientation. Behind the ball, the robot may not be too close to the ball (checkered area), unless it is facing it. This last rule is indicated by the arrows pointing in the direction of the ball. Approaching the ball is fine from any pose in the green area.

6.2.1 Action model accuracy

To evaluate the accuracy of this model, the robot executed another thousand runs, and compared predicted collision with observed collisions. The decision tree predicts collisions correctly in almost 90% of the cases. A more thorough analysis is depicted in the confusion matrix of the decision tree, in Table 6.1.

The model is quite pessimistic, as it predicts failure 61%, whereas in reality it is only 52%.

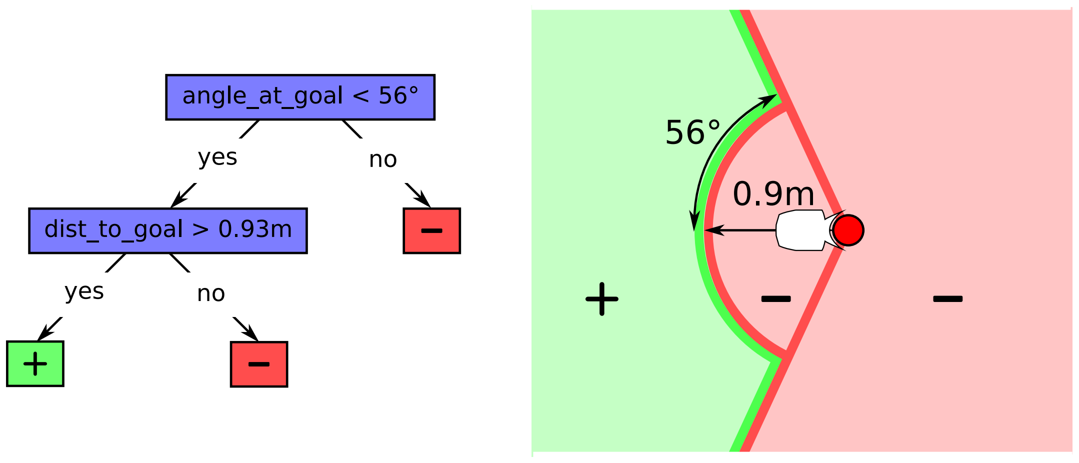


Figure 6.5. The learned decision tree that predicts whether an unwanted collision will happen.

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

Table 6.1. Confusion matrix for ball collision prediction. The model is correct in 89% of cases

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` was passed to the decision tree (Witten and Frank, 2005).

To learn an accurate model, it is necessary to gather as much as 1000 episodes for learning. Due to time constraints, this amount of data could not be gathered on the real robots in due time. Therefore the models and results in this chapter only apply to the simulated soccer robots.

6.3 Subgoal Assertion

This decision tree is exactly what we need to discern between the `Success` and `Fail` subsets. In a sense, it *is* the condition refinement. Whenever the decision tree returns `Success` for an initial state, the original action (`goToPose`) can be executed as is.

If the the action is predicted to fail, another action, that can satisfy the refined condition must be executed beforehand. In this case, this is simply another `goToPose` action. The intermediate subgoal for the first action can be any from the refined precondition. For the ball approach task, this is any position in the green area in Figure 6.5.

Although all positions in this area can function as an intermediate goal for the two `goToPose` actions, the overall expected execution duration is different for all off them. Of course, the subgoal with the smallest predicted execution duration should be chosen. In Chapter 5, we saw how such and optimization problem can be solved: with subgoal refinement!

Subgoal assertion was implemented whilst the implementation of the genetic algorithm was still underway, so the optimization has instead been done by random sampling. A thousand are samples randomly from the refined condition, and the predicted execution duration for both `goToPose` actions is computed and added. The subgoal with the minimal execution duration is then chosen to the the intermediate subgoal. As subgoal refinement was applied, the transitions at this subgoal is usually smooth.

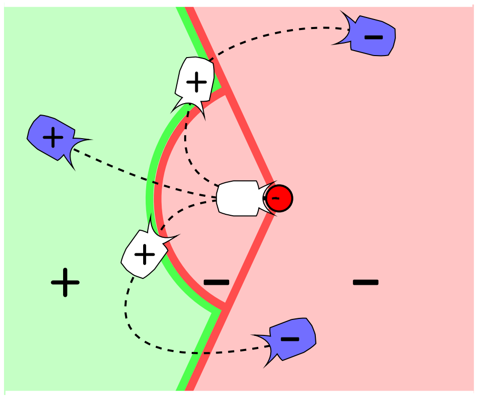


Figure 6.6. Subgoal assertion in the approach ball task

In Figure 6.6, three instances of the problem are depicted. Since the robot to the left is in the area in which no collision is predicted, it simply executes `goToPose`, without asserting a subgoal. The model predicts that the other two robots will collide with the ball when executing `goToPose`, and a subgoal is asserted. The optimal positions of the subgoals, determined by subgoal refinement, are shown as white circles.

The entire process of condition refinement, subgoal assertion and subgoal refinement can be encapsulated in a new abstract action, for instance `approachBall`. This process of encapsulating several action into one is known as “chunking” in architectures such as SOAR (Laird et al., 1986) or ACT-R (Servan-Schreiber and Anderson, 1990). Note that there is no executable action `approachBall`, as the executable `goToPose` is reused for this novel abstract action. Creating a novel abstract action is necessary however, as the pre- and post-conditions of `approachBall` have been refined as compared to those of `goToPose`.

6.4 Empirical Evaluation

To evaluate automatic subgoal assertion a hundred random ball approaches were executed in simulation, once with assertion, and once without. The results are summarized in Table 6.2. Before assertion, the results were, as can be expected, very similar to the results reported in Table 6.1. A collision is again correctly predicted approximately half the time: 52% in these hundred episodes, even slightly more than in the original 1000 episodes executed to obtain the action model. Subgoal refinement is applied in these cases, and is almost always successful: 50% of all episodes is transfered from having a collision to a successful ball approach. Only 2% of the episodes still have a collision, despite subgoal refinement. Because no subgoal refinement is applied when `Success` is predicted, there is no change in the lower prediction row. Consciously choosing not to apply subgoal refinement and not applying it are equivalent.

		Observed		Total
		`Fail`	`Success`	Predicted
Predicted	`Fail`	2% (=52%-50%)	60% (=10%+50%)	→ 62%
	`Success`	1%	37%	→ 38%
		↓	↓	
	Total Observed	3%	97%	

Table 6.2. Subgoal assertion results

Note that subgoal refinement was applied unnecessarily in 10% of episodes. In this case, both episodes with and without subgoal refinement were successful. However, the execution with subgoal assertion and consequent subgoal refinement was 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. Half of the time, a subgoal is introduced, 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.

Due to time constraints, subgoal refinement was not implemented or evaluated on the real robots, as no action model was learned. Instead, a failure model, similar to the one in Figure 6.5 was acquire by manually tuning the parameters of the model to a more cautious one. As subgoal refinements almost always chooses a subgoal somewhere on the border between the green and blue area in Figure 6.5, we wrote a heuristic that does the same. This `approachBall` action, although manually specified but still based on the learned model and subgoal refinement, was used for the real robots.

6.5 Related Work

6.5.1 Transformational planning

Sussman was the first to realize that *bugs* in plans do not just lead to failure, but are actually an opportunity to construct more robust and improved plans (Sussman, 1973). Although this research was done in the highly abstract symbolic blocks world domain, this idea is still fundamental to transformational planning.

XFRMLearn is framework in which human-specified declarative knowledge is elegantly combined with robot-learned knowledge (Beetz and Belker, 2000). Navigation plans are optimized with respect to execution time by analyzing, transforming and testing structured reactive controllers (SRCs) (Beetz, 1999). Designers first specify rules for analyzing and transforming these plans, and the robot then learns from experience when these rules should be applied. A substantial improvement in execution time of up to 44% is achieved. The analysis phase has many similarities with condition refinement, and transformation phase with subgoal assertion. 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 learns how to adapt to changing action requirements, such as refined goals.

6.5.2 Learning preconditions, effects and action failures

Methods for learning preconditions, such as the method presented in this chapter, can be summarized well by the following quote: “The problem of learning the preconditions for an action model can be viewed as a problem of concept learning in which the learner is given instances of action success or failure, and induces a concept describing the conditions which apply in successful instances.” (Shen, 1994).

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 and Amir, 2006). However, neither symbolic examples nor a symbolic precondition suffices to encapsulate the complex conditions that arise from the robot dynamics and its action parameterizations.

(Schmill et al., 2000) presents a system in which non-symbolic planning operators are learned from 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) proposes a method for learning the success rate of passing action in the context of RoboCup simulation league. Here a neural network is trained with 8000 examples in which a pass was 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 $>$70%”. This is also a good example of integrating human-specified and learned knowledge in a controller.

In (Fox et al., 2006b), an extension of the work in (Fox et al., 2006a), robots use learned action models to determine when an action is failing. The action model is learned by first mapping raw sensor data to observation by feature detection and classification techniques, then mapping observations to evidence items with Kohonen networks, and evidence items to states with state splitting (Fox et al., 2006a). This approach is used to learn a model of a robot that takes panoramic images by turning on the spot and halting at fixed intervals to take pictures.

With this Hidden Markov Model of the action, 50 training runs are generated. At each time-step, the log likelihood of the sequence of states given the learned model is computed. This yields 50 monotonously decreasing traces through time/likelihood space. The range of all these traces is defined to be normal behavior. During testing, failures are induced such as blocking the robot, or disconnecting communication. In three out of four error types this leads to traces that fall outside the range of the normal behavior, and an error is correctly recognized.

The emphasis in this work is not on predicting the failure of an action in advance, but rather recognizing when an action that is being executed is in the process of failing, as the following quote shows: “Planners reason with abstracted models of the behaviors they use to construct plans. When plans are turned into the instructions that drive an executive, the real behaviors interacting with the unpredictable uncertainties of the environment can lead to failure.” Therefore, cannot be used for condition refinement, but rather for execution monitoring.

6.5.3 Inductive transfer

The transfer of knowledge from one learning task to the next has been well studied within the context of connectionist networks (Pratt and Jennings, 1996). Here, it is termed “learning to learn”, or “inductive transfer” (Großmann, 2001). Two well known examples of this approach are Explanation Based Neural Networks (EBNN) (Thrun and Mitchell, 1993) and Multi Task Learning (MTL) (Caruana, 1997).

In EBNN (Thrun and Mitchell, 1993), a neural networks learns the mapping f_i from input to target values in the training set. In addition, EBNN also learns a mapping to the slopes (tangents) of f_i at the examples in the training set. These slopes provides information on how changes of the input features will affect the network’s output, and can therefore guide the generalization of the training examples. This second slope network represents a model of the domain, and can be used as an inductive bias for learning novel tasks, with the same network structure. This substantially reduces the number of needed training examples for novel tasks.

Suppose a mapping from four inputs to three different tasks must be learned from examples. One approach would be to train three neural networks, one for each task. With this approach, each of the three networks must learn the mapping to the output from scratch. Similarities between tasks can therefore not be exploited. In MTL (Caruana, 1997), only one network, in this case with three outputs, is learned, as depicted in Figure 6.7. In this network, representations that are common to all tasks are learned in the input to hidden layer mapping, and task specific representations in the hidden to output layer. Because all training examples can be used to learn the common representation, learning is significantly faster than when using a single network for each task. Empirical results have verified this (Caruana, 1997).

In analogy to the approach in this chapter is the differentiations between common task knowledge, and specific task knowledge. The `goToPose` action can be considered as the common knowledge needed to complete both the navigation and ball approach tasks. The learned model (condition refinement) and subgoal assertion are the specific knowledge needed to adapt the `goToPose` action to the novel ball approach task.

Both EBNN and MTL use multi-layer perceptrons as representation, and the transfer of knowledge is based on this representation. Furthermore, the learning performance and ease of

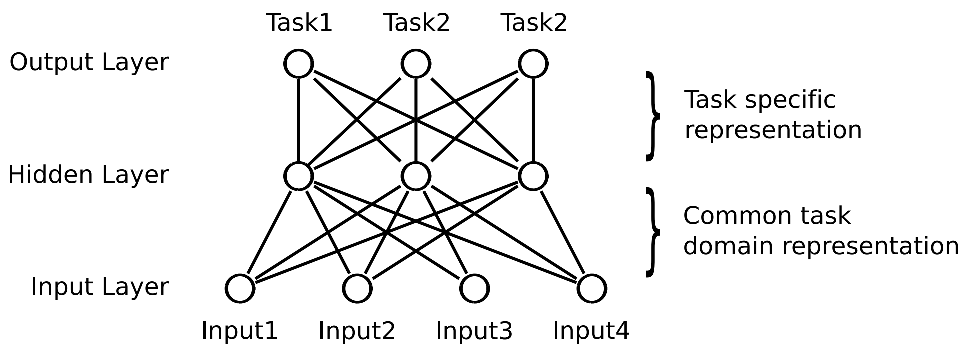


Figure 6.7. A multi task learning (MTL) network. Adapted from (Silver and Mercer, 1998).

transfer depend on the topology of the network, which is human-specified. Because MTL and EBNN depend on these a priori design decisions, they are only of limited use for autonomous learning (Großmann, 2001). For instance, they could not be applied to the task presented in this chapter, as it was not learned using a neural network. On the other hand, condition refinement and subgoal assertion, although their scope is limited to novel tasks with refined goals, could be used for tasks learned with neural networks.

6.6 Conclusion

In this chapter, we have presented condition refinement, which adapts pre-conditions to novel goal. These pre-conditions are learned with decision trees from observed experience, and are therefore grounded in the real world. Predicted failures can be resolved by introducing new subgoals, from which execution is predicted to succeed. In an interesting interplay between condition refinement and subgoal refinement, the best intermediate subgoal is chosen. We have demonstrated how the `goToPose` action could be reused to successfully approach the ball in the simulated soccer domain.

Condition refinement is a good example of combining common sense knowledge, which is provided by humans through the symbolic preconditions, with knowledge that the robots learn themselves. Also, condition refinement and subgoal assertion are important contributions to bridging the gap between symbolic planning, and plan execution on robots.

Directions for future work include the integration of condition refinement in an existing planning system, such as the one described in Section 5.2.1.

The results reported in this chapter have been published in: (Stulp and Beetz, 2006, 2005c).

Summaries of these publications are given in Appendix D.

7. Task Context: Multiple Robots

"Wat heb je nou liever? Één goed 11-tal of 11 goede 1-tallen?"
Johan Cruijff

As robotic systems are becoming more dextrous and sophisticated, they are capable of executing more complex tasks. Many of these more complex application tasks require two or more robots to cooperate in order to solve the task. A key aspect of these systems is that multiple robots share the same workspace, and can therefore not abstract away from the actions of other robots. The problem is how to tailor your actions in the context of actions of others.

Humans are very good at performing joint actions in shared workspaces. Consider two people assembling a bookcase (or a robot, as in Figure 7.1). With apparent ease, actions are *anticipated* and coordinated: one person holds a shelf while the other screws it in place, and so forth. A key aspect of this cooperation is that it is executed with little or no communication. Humans achieve this by inferring the intentions of others. Once the beliefs and desires of the cooperating party are known, we simply imagine what we would do in that situation. This is called the Intentional Stance (Dennett, 1987). If I see you grab a screw-driver, I can assume that you intend to screw the shelf in place; there is no need for you to tell me. By integrating your intentions into my own beliefs, I can also anticipate that my holding the shelf will ease our task, thereby coming closer to our joint desire of assembling the bookcase. Implicit coordination is used by humans in many domains: almost all team sports, construction of bookcases and others, and also in traffic.

Figure 7.1. Two humans implicitly coordinating the assembly of a Pioneer I robot.

In contrast, coordination in multi-agent and multi-robot systems is usually achieved by extensive communication of utilities. This is called *explicit* coordination. Previous work on cooperation seems to have focussed almost exclusively on this form of coordination (Botelho and Alami, 1999; Chaimowicz et al., 2002; Dias and Stentz, 2001; Parker, 1998; Werger and Matarić, 2000). It has also been used in the RoboCup mid-size league to allocate roles to the different players (Castelpietra et al., 2000; Spaan and Groen, 2002). However, implicit coordination has some important benefits over explicit coordination, related to:

Complexity To enable utility communication, protocols and arbitration mechanisms must be adopted between communicating entities , which adds complexities and can degrade the system. It is generally argued that communication can add unacceptable delays in information gathering and should be kept minimal (Tews and Wyeth, 2000).

Safety Because implicit coordination dispenses of the need for communication, there are many multi-robot domains that could benefit from this approach. Rescue robotics and autonomous vehicles operating in traffic are examples of domains in which robust communication is not guaranteed, but where correct coordination and action anticipation is a matter of life and death. When it comes to saving humans or avoiding accidents, it is better to trust what you perceive, than what others tell you: seeing is believing.

Human Robot Interaction Another recent research focus in which implicit coordination plays an important role is human-robot interaction, for instance in space exploration (Fong et al., 2005) or rescue robotics (Nourbakhsh et al., 2005). Our research group has a long-term project for human-robot interaction in intelligent rooms (Rusu, 2006). The room and robot are equipped with cameras, laser range finders and RFID tags, which provide robots with accurate information about what is going on in the room. When a robot and a human perform a joint action in their shared workspace, e.g. setting the table in the kitchen, or seam welding in outer space (Fong et al., 2005), it cannot be expected of humans to continuously communicate their intentions. Instead, the robot must be able to anticipate a human's intentions, based on predictive models of human behavior. We consider implicit coordination to be essential for natural interaction between robots and humans.

Mixed Teams In robotic soccer, there is an increasing incentive to play in mixed teams. Since robots in a mixed team usually have very different communication software and hardware, communication is often problematic. A solution would be to unify the software of the different robots of a potential mixed team. This would require substantial

> rewriting of at least one of the team's software. In our opinion this is undesirable. Why should an autonomous mobile robot have to commit to any kind of sensor processing or control paradigm to be able to cooperate with another team mate, if both are programmed to interact in the same problem domain? Professional soccer players certainly do not need to take a language course before being able to play soccer in a new country. Implicit coordination could solve the communication problem for robots in mixed teams by eliminating communication altogether.

A necessity for implicit coordination is being able to predict the outcome of the actions of others, by taking their perspective. As we saw in Section 3.2.1, it is hypothesized that the basis of social interaction and imitation in humans is also formed by forward models (Wolpert et al., 2003), as there are many similarities between the motor loop and the social interaction loop. It may be that the same computational mechanisms which developed for sensorimotor prediction have adapted for other cognitive functions. As we shall see, in implicit coordination, action models also enable robots to predict the performance of other robots.

In this chapter, we will apply implicit coordination to a typical coordination task from robotic soccer: regaining ball possession. Regaining ball possession is a goal for the team as a whole, but only one of the field players is needed to achieve it. The benefit of having only one player approach the ball is obvious: there will be less interference between the robots, and it also allows the other robots to execute other important tasks, such as strategic repositioning or man marking. Of course, the robots must agree upon which robot will approach the ball. The intuitive underlying locker-room agreement (Stone and Veloso, 1999) is that only the robot who is quickest to the ball should approach it. In Figure 7.2, implicit coordination has been highlighted within the overall system overview.

The next section will present the computational model of explicit and implicit coordination, and Section 7.2 will demonstrate how this model is applied to the ball interception task. In Section 7.3 we discuss some issues related to applying implicit coordination to heterogeneous teams. In the empirical evaluation in Section 7.4, three experiments, partially conducted together with members from the Neuroinformatics Group at University of Ulm, are presented. We conclude with related work and a summary in Sections 7.5 and 7.6 respectively.

7.1 Computational Model

In Figure 7.3, the computational model of explicit coordination is depicted. Vail and Veloso (2003) informally describe a similar methodology. Through a certain communication channel, the robot receives the utilities of other robots with respect to the task and possible actions at

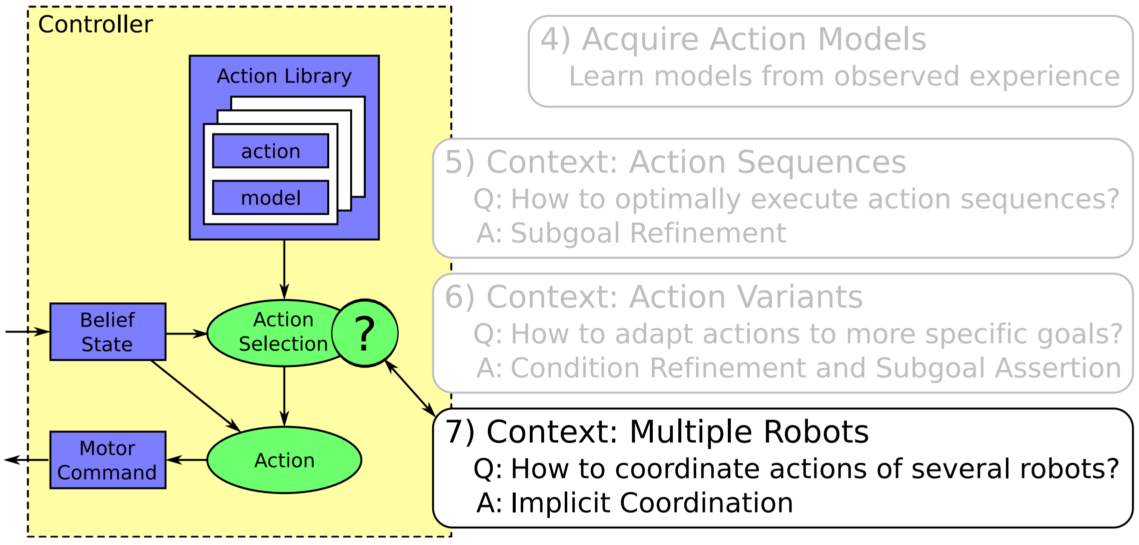

Controller
Action Library
action
model
Belief State
Action Selection
?
Motor Command
Action
4) Acquire Action Models
Learn models from observed experience
5) Context: Action Sequences
Q: How to optimally execute action sequences?
A: Subgoal Refinement
6) Context: Action Variants
Q: How to adapt actions to more specific goals?
A: Condition Refinement and Subgoal Assertion
7) Context: Multiple Robots
Q: How to coordinate actions of several robots?
A: Implicit Coordination

Figure 7.2. Implicit coordination within the system overview.

hand. The Joint Utility model then determines what the best action is, given the utilities of all robots.

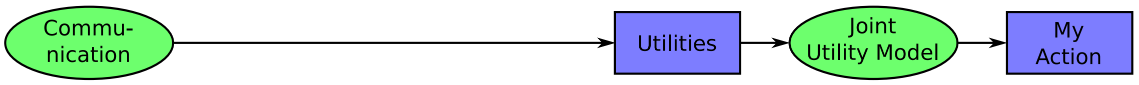


Figure 7.3. Explicit coordination, in which the utilities of other robots are communicated. This is the standard approach in robotics.

Implicit coordination, depicted in Figure 7.4, is a variation of explicit coordination, in which the utilities of others are not communicated, but computed by the robot itself. It does so by taking the perspective of others based on the states of others, and utility prediction models.

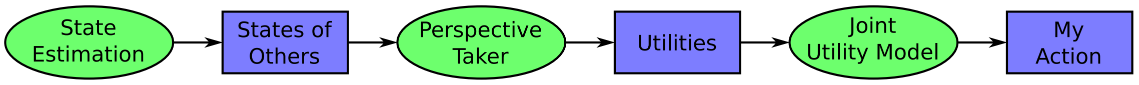


Figure 7.4. Implicit coordination without communication, in which utilities are computed from states using action models. States are either perceived. Humans use this approach to coordinate.

7.2 Applying Implicit Coordination

Here, the concepts used in Figure 7.3 will be explained using examples from the ball interception task, more or less from back to front.

My Action This is the action that the robot decides to execute. It should be coordinated with the actions of other robots. When regaining ball possession, this means that only one robot should approach the ball. This avoids interference between robots, and enables the robots that are not approaching the ball to perform other tasks, such as man marking or other offensive positioning.

Utilities In the ball interception task, the utility is approach time. The faster a robot can approach the ball, the higher the utility. This utility can therefore be computed by determining the execution duration of the `approachBall` action, given the current belief state. This time in its turn can be acquired by calling the learned action model for execution duration of the `approachBall` action, given the state of the robot and the ball. How this model is acquired has been extensively discussed in Chapter 4.

Joint utility model The joint utility model formalizes the intuitive rule that only one robot should approach the ball. It computes the best action a robot can execute, given its own utility for this action, as well as the utilities of other robots. So, for the ball interception task, the joint utility model returns `approachBall` if a robot predicted to be the fastest to approach the ball, and another action otherwise. For this task, the joint utility model therefore needs to know the expected time it will take to approach the ball for all robots.

Note that in this computational model, the joint utility model selects an action. For integration in plan-based control, the joint utility model could instead return a symbolic goal. The planner then determines an action sequence to achieve this goal.

Of course, all soccer teams will have implemented this strategy in some way, to avoid all robots continuously pursuing the ball. The contribution of the approach presented here is not to implement the concept of having only one robot going there. It rather shows how exploiting action models to reason about the outcome of the actions of others enables robots to become more independent of communication for coordination.

Communication In explicit coordination, robots compute only their own utility locally. It then sends its utility to the other robots, and receives the utilities of the other robots, over some communication channel. In auction based approaches (Gerkey and Matarić, 2003), the utilities are sent to a single arbitrator, which communicates roles or actions back to the robots.

Perspective-taker In implicit coordination, each robot computes the utility of all robots locally, without communicating the utilities. The perspective-taker enables each robot to make this prediction with respect to the current task and belief state. To do this, the robot swaps its own state with that of another robot in the belief state, and computes the utility. This "perspective-taking" (Trafton et al., 2005) is performed for all other robots, until the utilities for all robots are known. To compute the utility of others, the perspective taker computes the execution of the `approachBall` action for each robot. To do so, it needs to know the state and `approachBall` action model of each robot.

States of others As we saw, the robot needs to know the state of another robot to be able to take its perspective. In the belief state of the soccer robots, states are represented by a pose: the position and orientation of the robot. The states of others are determined through the state estimation module.

7.2.1 Utility vs. belief communication

The most difficult aspect of implicit coordination is estimating the states of others. Especially for robots with a limited field of view, such as ours, this is problematic. Therefore, we resorted to the communication of beliefs as a complement of state estimation, to acquire a shared and coherent representation, as depicted in Figure 7.5.

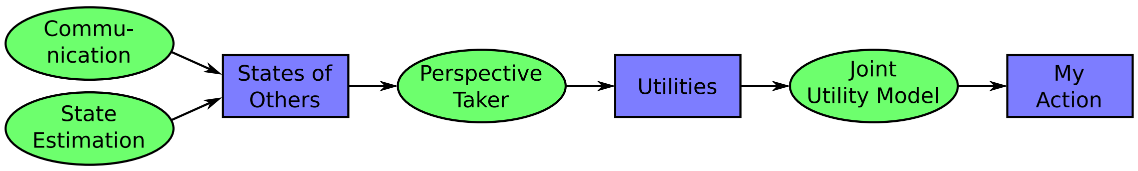


Figure 7.5. Implicit coordination with belief communication (BC).

This computational model might seem contrary to our communication-free paradigm, but there is an important difference between communicating *utilities* and communicating *beliefs*, which we shall explain in this section. Of course, implicit coordination without communication is the ideal situation, which we cannot achieve due to limitations in sensors and state estimation. Still, implicit coordination with state communication is preferable over explicit coordination for the following reasons:

- Since explicit coordination is only possible if you know the utilities of others, delays or failures in utility communications will often cause complete coordination failure. With implicit coordination, the robot can still rely on it's own sensors and state estimation to deduce the utilities of others. Coordination might then not be perfect, due to sensor limitations, but at least it does not collapse completely. One of the experiments in the experimental evaluation will verify this (Q6 in Section 7.4.2). In a sense, combining the two methods exploits the best of both worlds.

- Improvements in sensor technology and state estimation methods will allow robots to autonomously acquire a increasingly complete and accurate estimation of the states of others. In RoboCup for instance, almost all mid-size teams have resorted to omni-directional vision to achieve exactly that. So, beliefs needed to infer the utilities of others are becoming more complete and accurate, independent of communication. More accurate state estimation essentially replaces communication. Teams that have omni-directional vision could probably abandon communication altogether when using implicit coordination. This is certainly not the case for explicit coordination, which will always fully rely on communication.

- To enable human-robot cooperation, robots will at some point have to rely on state estimation only, as humans cannot be expected to compute their state. Implicit coordination with belief communication is an intermediate step to this ideal situation.

Summarizing, the robots use communication as a backup system if they cannot recognize the intentions of others, rather than as the backbone of their coordination. Improvements in sensor and state estimation will therefore allow implicit coordination to depend less and less on belief communication. This is necessary to simplify communication schemes, increase coordination robustness, and enable human-robot cooperation. This work proposes a step in this direction.

7.3 Implicit Coordination in Heterogeneous Teams

Due to scientific as well as pragmatic reasons, there is a growing interest in the robotics field to join the efforts of different labs to form mixed teams of autonomous mobile robots. For many tasks, a group of heterogeneous robots with diverse capabilities and strengths is likely to perform better than one system that tries to encapsulate them all. Also, for many groups, the increasing cost of acquiring and maintaining autonomous mobile robots keeps them from forming a mixed team themselves.

Therefore, the AGILO RoboCuppers have formed a mixed team with the Ulm Sparrows (Utz et al., 2004). The Ulm Sparrows (Kraetzschmar et al., 2004) are custom built robots. Their sensor suites consist of infrared based near range finders and a directed camera. The available actuators are a differential drive, a pneumatic kicking device and a pan unit to rotate the camera horizontally (180^o). One of the robots is depicted in Figure 1.4(c). As almost all robots in this league, the robots are custom built research platforms with unique sensors, actuators, and software architectures. Therefore, forming a heterogeneous cooperative team presents an exciting challenge. In the next sections, we will discuss the modification necessary to enable implicit coordination in heterogeneous teams.

7.3.1 Action models

When applying these models on-line in a game situation, the robots must know which player has which hardware platform to apply the correct model. To do so, each robot must have all models learned for all robots on the field, as well as a mapping from player number to temporal prediction model. This is implemented off-line.

Learning action models, in this case model trees that predict ball approach time, is no different for the Ulm Sparrows than it is for the AGILO RoboCuppers. Note that the action the Ulm Sparrows use to approach the ball is slightly different, as no orientation can be specified. Therefore, this action is called `goToPosition`. It took 40 minutes to gather the data for this model, and the accuracy of the learned model tree was already listed in Table 4.2.

7.3.2 Sharing belief in heterogeneous teams

To share beliefs, the teams must agree upon structures that encapsulate the information they want to exchange, and the communication framework over which this information will be sent.

The information in the belief state contains the dynamic pose of the robot itself, as well as the positions of observed objects, such as the ball, teammates and opponents. Each belief state message is accurately time-stamped, so that delays in communication can be registered.

The team communication uses a message-based, type safe high-level communications protocol (Utz et al., 2004). It is transfered by IP-multicast, as such a protocol keeps the communicated data easily accessible and prevents subtle programming errors that are hard to trace through different teams. As the communication in a team of autonomous mobile robots uses some kind of wireless LAN, that is notoriously unstable, a connection-less message based protocol is mandatory. With this approach, network breakdowns and latencies do not block the sending robot. IP-multicast is also used to save bandwidth, since this way each message has only to be broadcasted once, instead of m times for n clients.

The implementation uses the notify multicast module (NMC) of the Middleware for Robots (MIRO) (Utz et al., 2002). MIRO provides generalized CORBA based sensor and actuator interfaces for various robot platforms as well as higher level frameworks for robotic applications. Additionally to the method-call oriented interfaces, MIRO also uses the event driven, message-based communications paradigm utilizing the CORBA Notification Service. This standardized specification of a publisher/subscriber protocol is part of various CORBA implementations (Schmidt et al., 1997). Isik (2005) details about how MIRO has been ported to the AGILO robots.

Communicating the IDL-specified belief state discussed in at 10Hz with all teammates uses, on average, less than 10% of the available bandwidth of a standard 802.11b WLAN (11 MBit/s) (Utz et al., 2004). This should be available, even on heavily loaded networks, such as those in RoboCup tournaments.

7.4 Empirical Evaluation

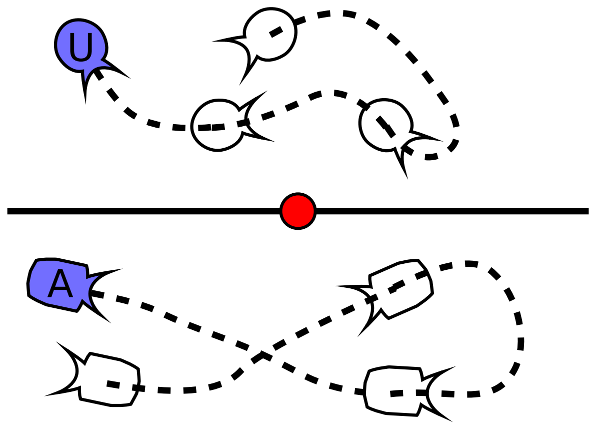
To evaluate the performance of applying implicit coordination in ball interception task, several experiments were conducted, first with three AGILO robots, and later with one AGILO and one Ulm robot.

7.4.1 Experimental design

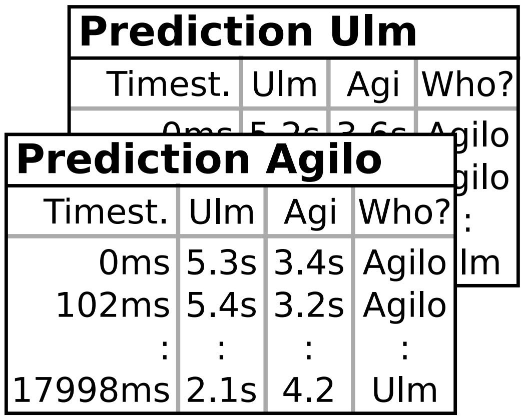
Three experiments were conducted, in a dynamic, static and simulated environment. The questions we will answer with these experiments are: Q1) Do the robots agree upon who should approach the ball? Q2) Do the robots choose the quickest one? Q3) Are temporal prediction models necessary, or would a more simple value such as distance not suffice? Q4) How robust is implicit coordination against errors in state estimation? Q5) When does implicit coordination fail? Q6) How do communication quality and state estimation accuracy influence coordination?

Dynamic environment experiment

This experiment was conducted with three AGILO robots, and in the mixed team with one AGILO robot and one Ulm Sparrow. In the experiments, the robots continuously navigated to randomly generated positions on the field. Once a robot reached its destination, the next random position was generated. These poses were generated such that interference between the robots was excluded, as depicted in Figure 7.6(a). For about half an hour (18 000 examples), the robots perform their random navigation routines. Each robot records the state estimation results locally every 100ms.



(a) Random navigation without interference.



Prediction Ulm

Timest.	Ulm	Agi	Who?
0ms	5.3s	3.6s	Agilo
			gilo
			:
			lm

Prediction Agilo

Timest.	Ulm	Agi	Who?
0ms	5.3s	3.4s	Agilo
102ms	5.4s	3.2s	Agilo
:	:	:	:
17998ms	2.1s	4.2	Ulm

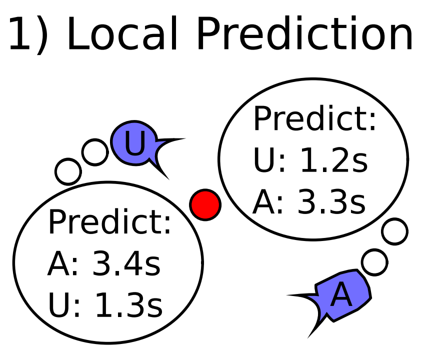
(b) Log-files collected in the dynamic experiment.

Figure 7.6. The dynamic experiment. The same experiment was also conducted with three AGILO robots.

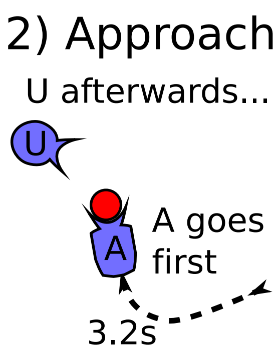
Figure 7.6(b) displays which information was gathered in each log file in the experiment with three AGILO robots. Apart from recording the temporal prediction for each robot, the robots also record who they think should approach the ball at that time, without ever actually approaching the ball. This allows much data to be recorded. Before the experiment, the robots synchronize their clocks. The times stamps can therefore be used to merge the three distributed files for further evaluation after the experiment.

Static environment experiment

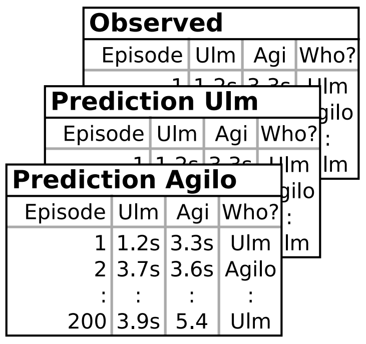
In the previous experiment, it is impossible to measure if the temporal predictions were actually correct, and if potential inaccuracies caused the robots' estimate of who is quickest to be incorrect. Even if robots always agree on the same robot, this is of little use if the robot is not indeed the fastest. Therefore a second experiment was conducted. During this experiment, the goal to approach is fixed. First, the robots navigate to random positions and wait there. They are then synchronously requested to record the same data as in the first experiment, but only for the current static state, as shown in Figure 7.7(a). Then, one after the other, the robots are requested to drive to the goal position, and the actual approach duration was recorded, see Figure 7.7(b). The log-files so acquired are almost identical to the ones in the dynamic experiment. The only difference is that they also contain the actual observed time for the robot. This static environment is less realistic, but allows the predicted time to be compared with the actually measured time for each robot.



(a) Step 1) Navigate to a random position, wait there. Record predictions.



(b) Step 2) Take turns approaching the ball and record observed result.



Observed

Prediction Ulm

Prediction Agilo

Episode	Ulm	Agi	Who?
1	1.2s	3.3s	Ulm
2	3.7s	3.6s	Agilo
:	:	:	:
200	3.9s	5.4	Ulm

(c) Log-files collected in the static experiment.

Figure 7.7. The static experiment. The same experiment was also conducted with three AGILO robots.

While executing this experiment, we realized a method to acquire the same data off-line. The two log-files were identical to the log-files gathered when learning the prediction model, as they also contain the current state, the goal state, and the real approach time. So, off-line, two samples from both temporal prediction log-files were chosen randomly, and added the predicted approach time for both robots. In order to do this, one sample of each pair had to be transformed, so that the goal positions of both samples coincide. This data is the same as we would have acquired during the experiment. In a sense, it is even more realistic, as the robot is moving in almost all samples, whereas it would have been static if the experiment had been conducted on-line.

Simulated experiment

Here, the experimental set-up is identical to the dynamic experiment. The simulator presented in Section B.2 in Appendix B allows us to vary two variables that most strongly influence the success of implicit coordination. The first is communication quality. At random times, and for random durations, communication is switched off in both directions. By controlling the length of the intervals, we can vary between perfect (100%) and no (0%) communication. The second is the field of view of the robot. We can set the view angle of the robot's forward facing camera between 0 (blind) and 360 (omni-directional vision) degrees. The other robot and the ball are only perceived when in the field of view. Gaussian noise with a standard deviation of 9, 25 and 22 cm is added to the robot's estimates of the position of itself, the teammate and the ball respectively. These correspond to the errors we have observed on the real robots (Stulp et al., 2004a). Since the dynamics of the Ulm Sparrows needed for simulation are not known, this experiment was only conducted with three AGILO RoboCuppers.

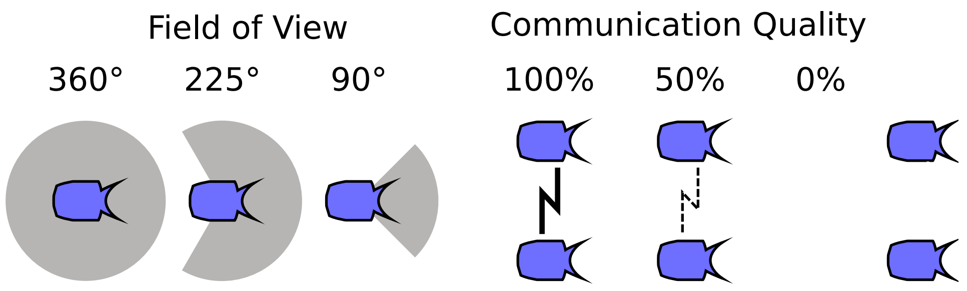


Figure 7.8. In the simulated experiment, the field of view and communication quality could be controlled. The experiment itself was identical to the dynamic experiment in Figure 7.6.

7.4.2 Q & A

Using the results of these experiments, we shall now answer the questions presented at the start of this section.

Q1) Do the robots agree upon who should approach the ball?

To answer this question, we simply determined how often all robots agreed on which robot should approach the ball. The results are listed in 7.1, in the row labeled "Chose the same robot?". Given the accurate estimates the robots have of each other's states, and the accurate predicted times that arise from this, it should not be surprising that the robots have almost perfect agreement (99% for agilo, 96% for the mixed team) on who should approach the ball.

	Action Model			Distance		
	Agilo		Mixed	Agilo		Mixed
Chose the same robot?	99%	Q1	96%	99%	Q3	95%
Chose the quickest robot?	96%	Q2	92%	81%	Q3	68%

Table 7.1. Accuracy of implicit coordination with belief communication

Q2) Do the robots choose the quickest one?

Agreeing about who should go to the ball is of little use if the chosen robot is not actually the quickest. Therefore, we would also like to know if the chosen robot is actually the quickest one to approach the ball. Of course, this could only be determined in the static experiment, in which the actual times it took each robot to approach the ball were recorded. A robot's decision to coordinate is deemed correct, if the robot that was the quickest was indeed predicted to be the quickest. In the experiment with three agilo robots, the robots were correct 96% of the time, and in the mixed team 92%, as can be seen in Table 7.1.

Q3) Are temporal prediction models necessary, or would a more simple value such as distance not suffice?

Using distance as a rough estimate of the approach time, as done in (Murray and Stolzenburg, 2005), would save us the trouble of learning action models. Although time is certainly strongly correlated with distance, using distance alone leads to significantly more incorrect coordinations. The last column in Table 7.1 shows this. Agreement is still very good (99%/95%), but the robot that is really the quickest is chosen only 81%/68% of the time. So, when using distance, the robots are still very sure about who should approach it, but they are also wrong about it much more often.

Q4) How robust is implicit coordination against errors in state estimation?

As we saw, almost perfect coordination was achieved in the dynamic experiment. This is not so surprising, as the robots have very accurate estimates of each other's states. To analyze how noise in the estimates of the other robot's states influences coordination, we took the original log files of the three AGILO robots, and added Gaussian noise of varying degrees to the estimates that robots have of each other's pose ($[x_t, y_t, \phi_t]$). The predicted times were then computed off-line, based on these simulated log files.

The results are shown in Figure 7.9. The x-axis shows the standard deviation of the Gaussian noise added to the data. So the first column, in which there is no added noise, represents the results of the dynamic experiment with the three AGILO RoboCuppers, which had been listed in Table 7.1. The y-axis shows the percentage of examples in which 0,1,2 or 3 robots intended to approach the ball. Of course, '1' means that coordination succeeded. This graph was only generated for the initial experiment with three AGILO RoboCuppers

We can clearly see that coordination deteriorates when robots do not know each other's states so well. If you have a robotic (soccer) team, and know the standard deviation between the robot estimations of each other's positions, the graph gives an indication of how well implicit coordination would work in this team.

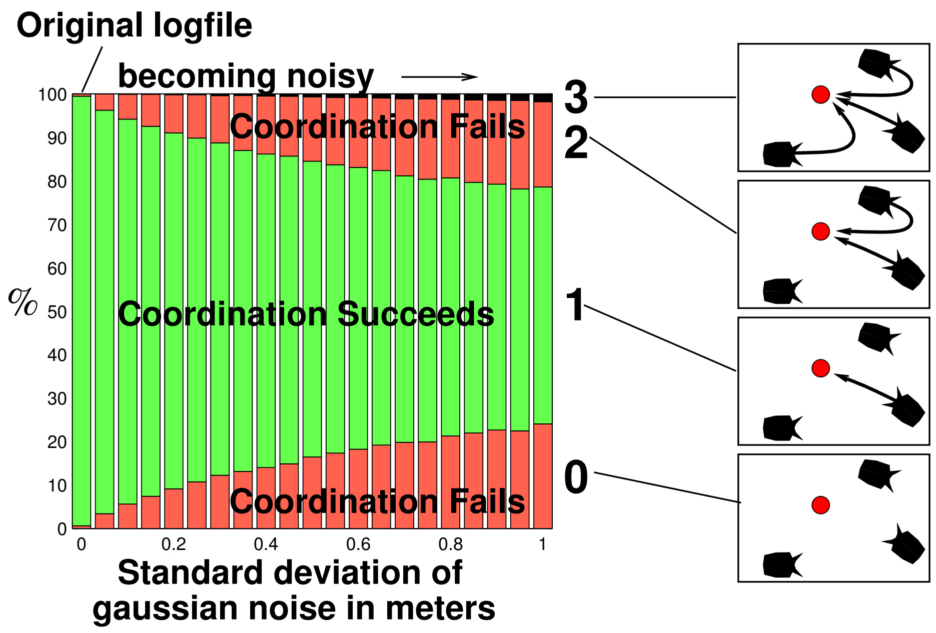


Figure 7.9. Influence of simulated state estimation errors on implicit coordination.

Q5) When does implicit coordination fail?

In the log files of both the mixed team and the AGILO only team, we labeled all examples in which exactly one robot decided to approach the ball with `Success`, and others with `Fail`. A decision tree was then trained to predict this value. The learned trees are represented graphically in Figure 7.10. For both prediction models the main rule is that if the difference in predicted times between two robots is small, coordination is likely to fail, and if it is large, it is likely to succeed. This is intuitive, because if the difference between the times is large, it is less likely that adding errors to them will invert which time is the smallest. Note that in between these two limits, there is a 'gray' area, in which some other rules were learned. They only accounted for a small number of example, so for clarity, we will not discuss them here.

In Figure 7.11, we present an illustration of how such failure prediction could be used in practice. It is easiest to understand this image if one imagines that the robots are standing still at the drawn positions, and the ball is rolling slowly from left to right. At every 5cm of the ball's trajectory, the robots determine who should approach the ball at that time, using implicit coordination. Note that the arrow does not represent the direction that the ball is rolling, but

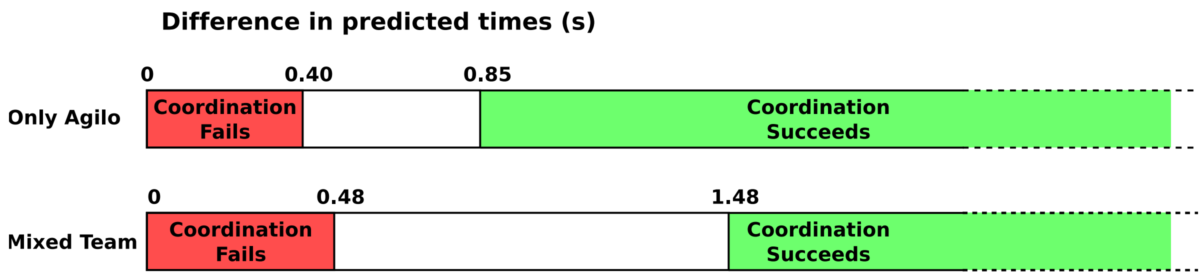


Figure 7.10. Representation of the decision trees that predict coordination success.

rather the direction from which the robots should approach it. After ball interception, their goal is to dribble it in this direction. The robot that is chosen to intercept it is connected to the current ball's position by a solid green line. When the decision tree predicts that coordination might fail, the robots between which confusion might arise are both connected to the ball's position by a red dashed line. Note that this image was generated in simulation, not with the real robots.

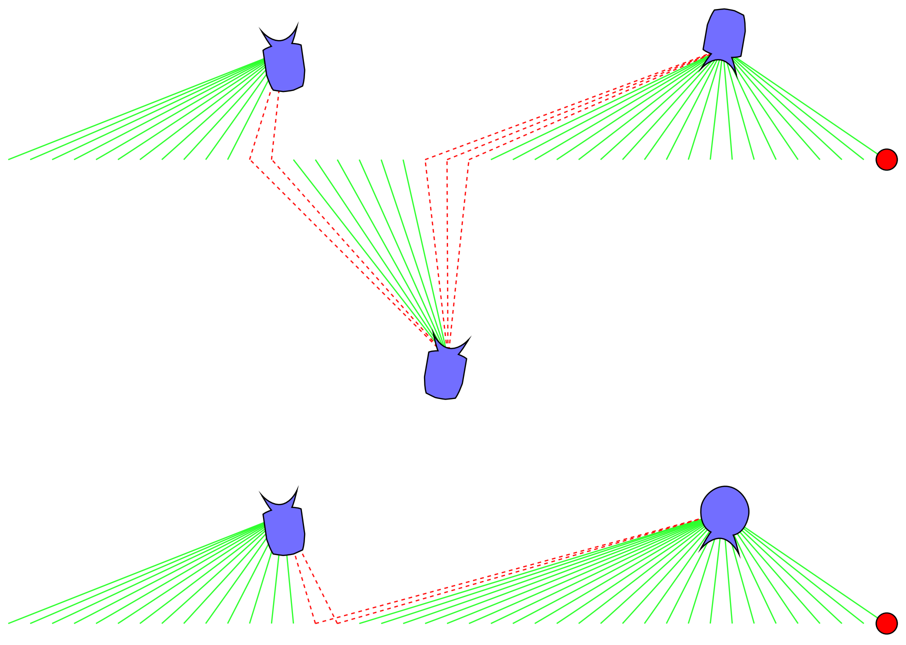


Figure 7.11. Example of implicit coordination with failure prediction. Solid green lines represent that only one robot would approach the ball at this position. Dashed red lines show when coordination is predicted to likely fail. The robots must all approach the ball from the right, as indicated by the arrow.

Humans also recognize when coordination might fail. For example, in sports like soccer or volleyball, it is sometimes not completely clear who should go for the ball. Humans solve this problem by making a brief exclamation such as “Mine!”, or “Leave it!”. So in these cases, humans resort to explicit coordination and communicate their intentions. Not only do humans have utility models of each other to coordinate implicitly, they are also aware when confusion might arise. The learned decision tree essentially provides the robots with similar awareness, as they predict when implicit coordination failure is likely. So, they could be used to determine when robots should resort to other methods of coordination. For instance, soccer robots could have a simple locker-room agreement that when coordination failure is predicted, the robot with the higher player number should approach the ball (excluding the goalie).

Q6) How do communication quality and state estimation accuracy influence coordination?

The results of the simulation experiment, which show how the performance of different coordination strategies depends on the quality of communication and the field of view, are depicted in Figure 7.12. Communication quality is the percentage of packets that arrive, and field of view is in degrees. The z-axis depicts coordination success, which is the percentage that only one robot intended to approach the ball. The computational models of the different forms of coordination have been repeated below these graphs.

Since explicit coordination is based completely on communication, it is not surprising that it perfectly correlates with the quality of the communication, but is independent of the size of the field of view. No communications means no coordination, and perfect communication means perfect coordination. For implicit coordination without communication, the relation is converse. If a robot is able to estimate the states of others better, it is able to coordinate better. The third graph shows implicit coordination with belief state exchange (as used on our real robots). If the robot has another in its field of view, it determines the other’s state through state estimation, otherwise it uses communication (if possible) to exchange beliefs. These states are then used to predict the utilities of others, independent if they were perceived or communicated.

This graphs clearly verify the hypothesis from Section 7.2.1 that implicit coordination with belief exchange achieves better performance with communication loss than explicit coordination alone. Instead of complete coordination failure in case of communication loss, there is a graceful decay, because a second system based on state estimation can still be used to estimate the utilities of others. In Section 7.2.1, we also hypothesized that improvements in sensors and state estimation would allow robots to acquire more accurate and complete belief states, and rely less on communication for coordination. The arrow in the third graph in Figure 7.12 represents this direction.

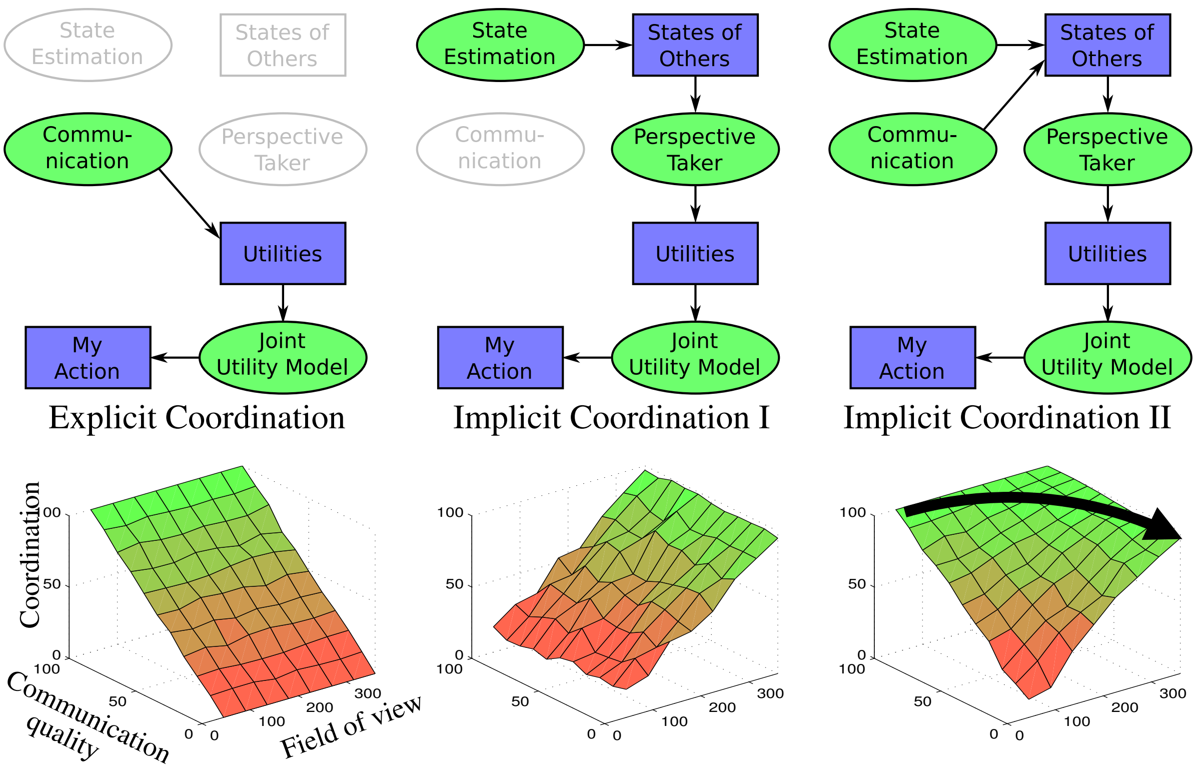


Figure 7.12. Results of the simulation experiment, which show how the performance of co-ordination strategies depends on the quality of communication and the field of view.

7.5 Related Work

7.5.1 Explicit and implicit coordination

Previous research on cooperation has focussed almost exclusively on explicit coordination (Gerkey and Matarić, 2003). On the other hand, work on implicit coordination usually assumes that all agents have access to a central and global representation of the world, which is enabled by simulation, as in (Sen et al., 1994), or global perception, as in the RoboCup small-size league (Tews and Wyeth, 2000; Veloso et al., 1999). In all this work, teammates are not reasoned about explicitly, but are considered to be mere environment entities, that influence behavior in similar ways to obstacles or opponents.

Stone and Veloso (1999) deals with the issue of low band-width communication in the simulation league is by *locker-room agreements*, in which players agree on assigning identification labels to certain formations. During the game, only these labels, instead of complete formations, must be communicated.

Murray and Stolzenburg (2005) combines implicit and explicit coordination to achieve ball approach coordination in the simulation league. First, each robot determines the distance of each teammate to the ball. Based on this, each agent decides if it will approach the ball or not. Coordination is still explicit, because the agent who decides to approach the ball first must ‘lock’ a shared resource, which prevents other robots from chasing after it. The use of this global resource requires communication.

Most similar to our work is (Vail and Veloso, 2003), in which robots in the legged-league also coordinate through implicit coordination which is based on representations which are completed through the communication of belief states. Communication is essential, and assumed to be flawless. It is not investigated how communication loss influences coordination. The utility measure is a sum of heuristic functions, which are represented as potential fields. Whereas our utility models are grounded in observed experience, and have a well-defined meaning (e.g. execution duration in seconds), these heuristic functions have no clear semantics. Therefore, customizing these functions to individual robots is difficult, as the semantics of and interactions between them are not fully understood. However, this customization is essential for achieving efficient coordination in a heterogeneous team with robots with different dynamics and capabilities.

Buck et al. (2002b) describes a method in which robots are also coordinated by predicting approach times locally. The motivation behind this work is that a framework for communicating state was already available, and using implicit coordination with action models was simply easier to implement than novel utility communication and arbitration modules. The research in this chapters extends this work by making a comparison of explicit and implicit coordination, learning models of when coordination fails, and enabling coordination in heterogeneous teams.

7.5.2 Heterogeneous teams

The idea of cross team cooperation has some tradition within the RoboCup leagues. In the simulation league, the source code of many teams was published on the Internet allowing new participants to base their new team on previous participants of simulation league tournaments.

The most similar mixed team cooperation effort was the Azzurra Robot Team, a mid-size team from various Italian universities. They also used a (proprietary) publisher/subscriber communication protocol, utilizing UDP. This team used explicit coordination (i.e. with utility communication) to assign roles among the field players (Castelpietra et al., 2000). Unfortunately the Italian national team was dissolved after the RoboCup tournaments in 2000.

One of the most successful mixed teams in RoboCup has been the GermanTeam, which participates in the legged-league (Röfer, 2002). The GermanTeam is a cooperation of five universities participating with one team and one code repository. The exchange and integration of software is enabled by a standardized hardware platform, as well as a modular software design. The challenge we face is to integrate different hardware systems and software architectures, for which integration has never been a primary goal. A bottom-up design, such as the GermanTeam has, would require complete rewrites of all systems, so instead we have chosen a software package that extends each individual software architecture.

Many RoboCup teams acquire coherent and complete beliefs by communicating and sharing their belief states. The use of shared representations was probably one of the key reasons for the success of the Freiburg mid-size team (Dietel et al., 2002).

7.6 Conclusion

Whereas humans coordinate with little or no communication, robots usually rely on extensive communication of utilities or intentions. In this chapter, we have implemented a framework that enables robots to reason about the utilities of others in a ball interception task, and coordinate their global behavior by making only local decisions, based on the action models and states of the other robots. Unfortunately, the state estimation is not reliable enough to accurately and robustly determine the states of others, so it is necessary to communicate belief states. We have motivated why state communication is preferable over utility communication. The robustness of implicit coordination was demonstrated in both a homogeneous and heterogeneous team of soccer robots.

We have shown that action models outperform more simple performance measures such as distance, and that action models can be learned for robots of other teams. Due to the redundancy in using both state communication and estimation, implicit coordination is more robust against network failures, which was evaluated in several experiments. These aspects must be taken into account when transferring multi-agent research to multi-robot teams. This chapter is a contribution to the evaluation of advantages and disadvantages of implicit and explicit coordination in robotic teams.

Future work includes learning action models for opponent robots. If the actions of the opponent could be anticipated, a robot could *coordinate* its own actions with that of the opponents. Of course, this coordination is only beneficial for one of the robots. Furthermore, we have done some preliminary work on learning temporal prediction models that take opponent robots into account. Due to the increased state space, and the unpredictability of what the opponent will do, learning accurate models is more complicated. By partitioning all possible scenarios into several classes, relatively accurate models could be learned for each class.

The results reported in this chapter have been published in: (Stulp et al., 2006a; Stulp and Beetz, 2006; Isik et al., 2006; Utz et al., 2004; Stulp and Beetz, 2005c). Summaries of these publications are given in Appendix D.

8. Conclusion

To adapt to novel environments and tasks, agents must be able to learn. Learning means experimenting, observing the results of experimentation, and generalizing over that which was observed. Forward models, which predict the outcome of motor commands, are good examples of knowledge that humans learn from experience, and use to adapt to novel contexts. The concept of a forward model can be extended to action models, which predict the outcome of durative actions. We have shown how robots can acquire such action models.

On the other hand, domain knowledge formalization as well as abstraction and reasoning capabilities are currently not yet at a stage that enables robots to robustly acquire declarative common-sense knowledge autonomously. Therefore, it is common that such knowledge on *what* to do in the first place is specified by human controller designers. The key idea in this dissertation is to merge this human specification with learned action models, as they complement each other well.

To do so, we developed a framework in which action models are integrated in a controller, partially specified by human designers. The action models enable the robot to autonomously answers questions that designers find difficult to answer themselves, even for their own actions. We have implemented several applications of action models, with an emphasis on answering questions that arise when applying existing actions to novel task contexts:

- Subgoal refinement optimizes action sequences with partially specified subgoals, by extracting free action parameters, and optimizing them with respect to the expected performance, predicted with action models. The resulting motion is more efficient and fluent.
- Condition refinement and subgoal assertion, in which preconditions are refined by learning when executing an existing action will succeed at achieving novel goals. Failure prediction is resolved by introducing intermediate goals, which are optimized with subgoal refinement.

- Implicit coordination enables robots to coordinate their actions by reasoning about the utilities of others, using action models and knowledge about the states of others. Coordination that relies on state estimation and communication is more robust than relying on communication of utilities alone.

We have demonstrated that enabling robots to refine and improve their actions and plans *themselves* not only alleviates the designer's task, but also improves the robot's performance, autonomy, adaptivity and robustness. Robots can only do so if they learn to predict the outcome of their actions from experience, as we do ourselves.

Appendices

A Action Libraries

A.1 Action: `goToPose`

This is a navigation action that takes the robot to a target position with a target orientation and speed, and returns the desired translation and rotational velocity. It is implemented by computing an intermediate position behind the goal pose, where behind is defined in terms of the orientation at the desired pose. This intermediate position (IP) behind the desired pose is then approached. As the robot closes in on the IP, the IP approaches the final goal pose, thus luring the robot towards the desired position. Since the robot initially approaches the goal pose from behind, it is has the correct orientation one the goal pose is reached. Behnke and Rojas (2001) outlines a very similar method. Some example runs of this action will be visualized in the next chapter, in Figure 4.2.

This navigation action was used on the AGILO robots previously with the Pioneer 1 controllers. With different parameterizations, it could also be used for the AGILO robots with the Roboteq controllers, as well as the simulated B21.

A.2 Action: `goToPosition`

The Ulm Sparrow robot is from a different research group altogether. Therefore, we have no knowledge of how the `goToPosition` of this robot was implemented. The interesting aspect of learning and applying action models is that the implementation of the action need not be known, because the models are learned from observed behavior, not an analysis of the inner workings of the robot. However, it is necessary that action parameters are known, as the robot must known with which variables the action model should be learned and called. These are the same as for the `goToPose` actions, with the exception that the target orientation cannot be set. It also returns a motor command that contains desired translation and rotational velocity, thought this knowledge is also irrelevant for learning or applying action models.

A.3 Action: `reach` (B21)

The exact implementation of this action was also not known. It had been previously developed, and integrated in the B21 model in the Player module of the Player/Stage framework. For this reason, the exact representation of the motor command is not known. Again, the signature of the action was known, and listed in Table 2.1. The x,y,z coordinates specify the 3-D location of end of the arm relative to the robot body, and the ax,ay,az the angles of the gripper relative to the arm.

Again, the action parameters are all that is needed to acquire an action model. The same holds for humans. Although we have several inverse models (actions) to reach for objects, we are not aware that there are several of them, and find it difficult to explain exactly how we perform this action (Haruno et al., 2001). We simply do. Note that this does not keep us from learning forward models (action models) for these actions (Flanagan et al., 2003).

A.4 Action: `reach` (PowerCube)

In the PowerCube domain, the state is represented in joint space with the angles and angular velocities at both joints: $\theta^a, \dot{\theta^a}, \theta^b, \dot{\theta^b}$. The `reach` action on the PowerCube takes the arm from one state to the next using a ramp velocity-profile. The ramp has three phases: acceleration, cruise speed, de-acceleration. Each joint accelerates with a constant acceleration value, reaches the desired cruise speed and stays there until it begins the de-acceleration phase, which is done also with constant acceleration. The trajectories of both joints are synchronized so they begin exactly at the same time, and have the same length. This allows us to control the combined speed of the end-effector of the arm at desired states, by decomposing this speed and direction into the appropriate velocity for each joint. A PID controller sends power commands to the joints to allows fine control of the action.

B AGILO RoboCuppers: Hardware and Tools

In this appendix, the hardware of the AGILO RoboCuppers will be introduces, along with some of the tools used in controller development.

B.1 AGILO RoboCuppers hardware

The AGILO team is realized using inexpensive, off-the-shelf, easily extendible hardware components and a standard C++ software environment. The team consists of four customized ActivMedia Pioneer I robots (ActivMedia Robotics, 1998) (1); one of which is depicted in figure B.1. The robot has a controller-board (2) and differential drive (3). For ball handling, the robot has a passive ball guide rail (4) and a spring-based kicking device (5). The only sensor apart from the odometry is a fixed, forward-facing color CCD Firewire camera with a lens opening angle of of 90^o (6). All computation is done on a standard 900 MHz laptop with Linux operating system (7). The robot uses a Wireless LAN device (8) for communication with teammates (Stulp et al., 2004b).

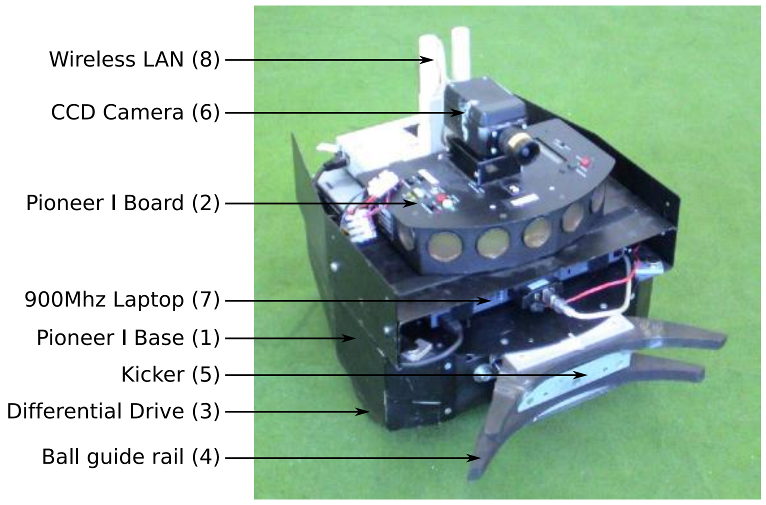


Figure B.1. The hardware components of the AGILO soccer robots.

During the research, we upgraded the controller boards from the original board delivered with the Pioneer I robot to the Roboteq AX2550 board (Roboteq Inc., 2004). Models have been learned for both robots. When discussing these robots we shall normally refer to the version with the novel Roboteq board, and explicitly mention when the original Pioneer I board was used.

B.2 Simulator

Robot simulation in general is a powerful tool for the development of autonomous robot control systems because it allows for fast and cheap prediction and makes experiments controllable and repeatable. The first step in developing or adapting skills for our robots is made in the MRose simulator (Buck et al., 2002a). The main features of the MRose simulator shows its focus on learning and designing controllers:

Accurate Dynamics The skills designed in the simulator can only be used on the real robots if the dynamics of the simulated robots is similar enough to that of the real robots. Therefore, the dynamics have been learned using neural networks, from experience observed on the real robots (Buck et al., 2002a).

Fast To learn actions and action models, sufficient experience needs to be available. To quickly gather sufficient data, it is essential that simulation is an order of magnitude faster than the real world time. The learned dynamics facilitate this, as well as simulating the robots in only two dimensions. These features enable the simulator to run at 100x real-time.

No State Estimation Sensors and state estimation are not part of the simulator. The inaccuracy and uncertainty that arise from sensing and state estimation are simulated.

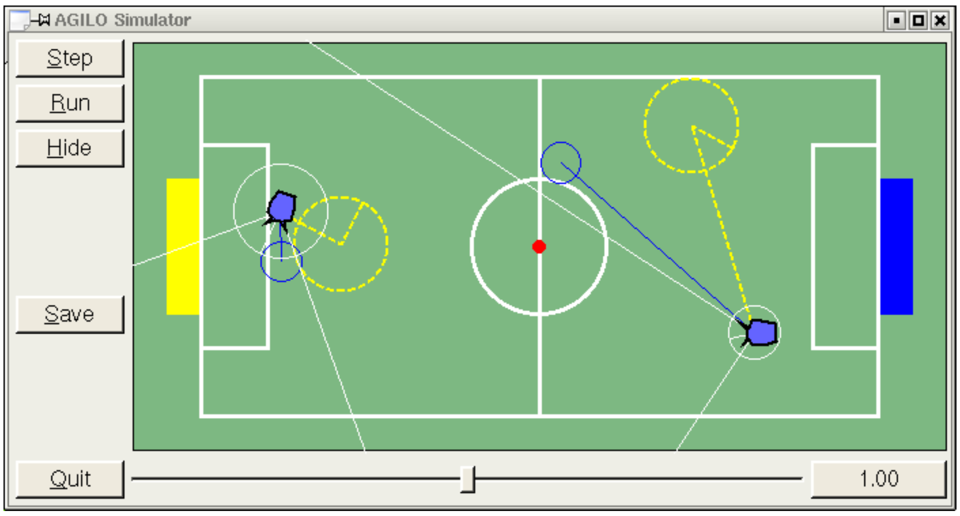


Figure B.2. The Qt simulator GUI.

We have equipped the physics engine of the MRose simulator with a new Graphical User Interface, written in Qt (Trolltech, 2005). This GUI allows the controller to visualize internal parameterizations in the field, as shown in Figure B.2. Here, the blue circle is the intermediate goal, and the yellow circle the final goal. Circle radius indicates the desired translational velocity. Such information is very useful for debugging. The slider below allows the simulation acceleration to be set. It can be set from 0.1x (slow motion), over 1.0x (to monitor real-time behavior) to 100x (to gather data) real-time. The field display can be turned off to have the simulated world to run at top speed. The simulator can also be started without the GUI, allowing many examples to be gathered in little time

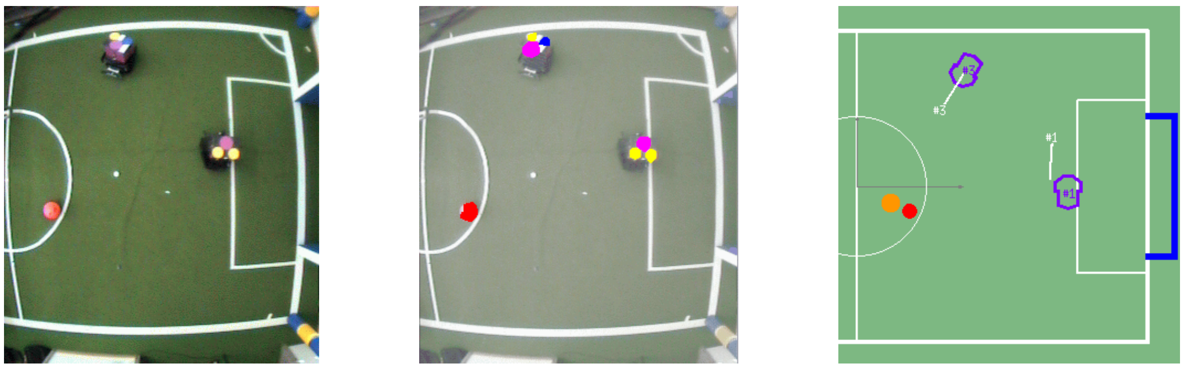
B.3 Evaluation with ground truth

Evaluating dynamic robotic multi-agent systems as in robotic soccer is difficult for several reasons. Since these systems are dynamic, it is difficult to capture the state of the world at a certain time or at certain time intervals without interfering with the course of events. How to accurately measure the position of a robot, if it is traveling at 2m/s? Robotic platforms usually suffer from noisy sensors and hidden state. A robot's beliefs about the world are therefore incomplete, uncertain and inaccurate. How to determine where a robot *really* was, if you only have its belief state to judge by? Multi-agent systems also require that several subsystems are evaluated at the same time, as well as the interactions between them. Furthermore, for the experiments presented later, it is important that the variables are controllable and reproducible.

For these reasons, we have used our ground truth system (Stulp et al., 2004a). This vision-based system can automatically provide *ground truth* about the state of the world in dynamic robotic multi-agent systems. It is very similar to the global view cameras use in the RoboCup small-sized league. It consists of one or more cameras mounted above the field looking downward. Each robot has a distinctive top-marker that is easy to detect by these cameras. Since the cameras are static, and can locate the markers precisely, this yields very accurate data on the location and orientation of each robot on the field.

The ground truth system consists of two cameras with an opening angle of 90°, at a height of approximately 3m above the field. The cameras are facing downward, and together they cover the whole training field, which is 6.4m x 10.4m. The robots can be distinguished from one another using color markers, exactly as in done in the RoboCup small-size league (F180 Laws, 2004). Each camera grabs images at a rate of 15Hz. The first image in Figure B.3 shows an example of such an image. The images are then segmented by color using the look-up tables generated during color calibration, as the center image of Figure B.3 shows. The acquired blobs are then filtered according to size and shape. With the configuration of blob groups, the position, orientation, team and player number of each robot can be determined.

This information is logged in a log-file, together with the belief states of the other robots. It



(a) Original image.

(b) Relevant blobs.

(c) Monitor view.

Figure B.3. Intermediate steps in ground truth image processing..

can also be communicated to the robots themselves, as well as the program uses to monitor and display the state of the world, as can be seen in the last image Figure B.3. In this example, there are two robots, whose self-localization is displayed in blue. Their actual position, determined by the ground truth system, is displayed as a white line, the start of which indicates the robot's center. The orange ball is where robot 3 beliefs the ball to be, and the ground truth position is displayed in red. This graphical display allows us to make quick on-line inferences: "Robot 3 is localized very well, and has localized the ball reasonably. Robot 1 is not localized that well, but good enough for performing useful behavior."

To determine the accuracy of robot localization by the ground truth system, we placed a robot with marker on fifteen different positions on the field. We measured the actual position by hand (*ground* ground truth, so to speak), and compared it to the pose estimated by the system. For the localization of the robots we have an accuracy of 0.3 to 5.2 cm and for its orientation 1 to 2.3°. Apart from the accuracy, another important issue is whether a marker is detected at all. Three experiments, described in (Stulp et al., 2004a), were conducted to determine the robustness of marker detection. In a static environment, the number of false positives is only 0.1%, and the number of false negatives is 1%, averaged over all eight markers. This last value is 2.5% in dynamic environments.

Providing robots with the global state

Having access to the global game state also allows a thorough evaluation of the action selection module, independent of the inaccuracies and uncertainties that arise from the state estimation performed locally on the robot.

In our system, the first step in developing or adapting control routines is made in the MRose simulator (Buck et al., 2002a). This simulator has an accurate learned model of the robot dynamics, and can simulate multiple robots on one field in parallel, using the same controller the robots use in the real world. Even though this simulator has good models of the environment, the low-level routines do not map to the real controller perfectly. Testing of the controller on the real robot is necessary to fine-tune the low-level routines. Without ground truth, this is difficult, as the robot's imperfect state estimation makes is difficult to see the effects of changes to the low-level controllers, because unexpected behavior might arise due to false self-localization.

To make this process easier we have enabled functionality to provide the robots with the global state, as computed by the ground truth cameras. This is exactly the same as in RoboCup small-size league. Using this set-up, we can test the robots' control routines, without depending on state estimation.

The topics reported in this section have been published in: (Beetz et al., 2004; Stulp et al., 2004a,b,c). Summaries of these publications are given in Appendix D.

C Tree-based Induction

C.1 Decision Trees

A decision tree is a flow-chart-like tree structure, in which internal nodes denote a test on an attribute, a branch represents an outcome of the test, and the leaf nodes represent class labels or class distributions. The famous decision tree example from the textbook “Artificial Intelligence: A Modern Approach” is depicted in Figure C.4. An example set of attributes can be classified by traversing the tree, choosing branches based on the attributes in the example and the test in the nodes, until a leaf is reached. The class in this leaf is the classification for this set of attributes. In the example, the waiting for a table is decided on evaluating the attributes `Patrons?`, `WaitEstimate?`, etc, until one of the decision leaves `Yes` or `No` is reached.

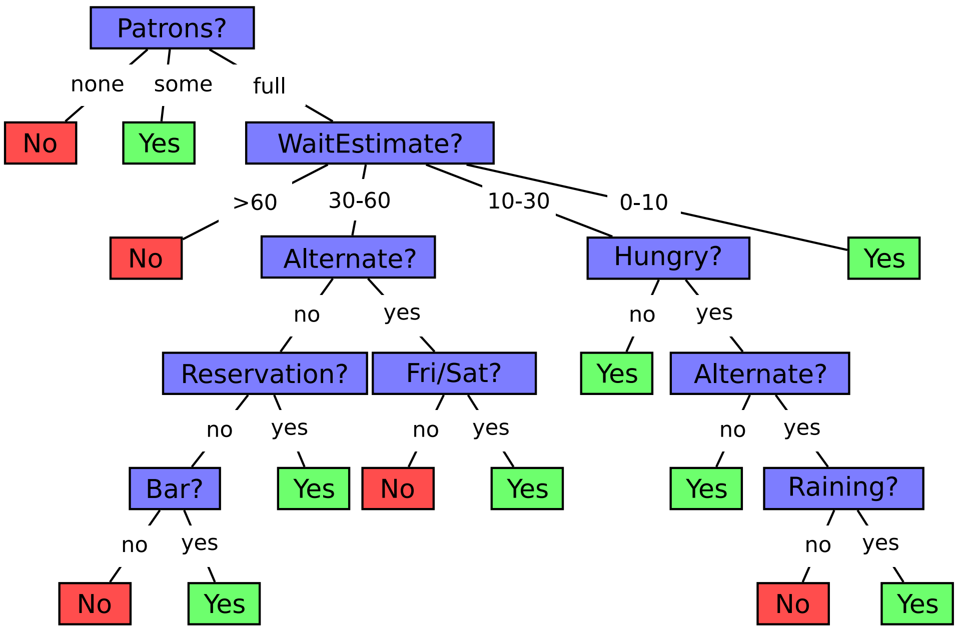


Figure C.4. A decision tree for deciding whether to wait for a table. Adapted from (Russell and Norvig, 2003).

Decision trees can be learned from a set of examples, which consist of specific values assigned to the attributes, along with the value of the target class. The decision tree is induced by a process known as recursive partitioning. At the start, all the training examples are at the root. A certain attribute is then chosen, and the examples are partitioned into n sets, one for each of the n values the attribute can take. In each set, all examples have the same value for the chosen attribute. This partitioning continues recursively on the set in each node, until all or most examples at each node have the same target value.

The first issue in decision tree induction is which attribute to use to partition a set of examples. The ideal attribute would separate the examples into *pure* sets in which each example has the same target class. Because such an ideal attribute is often not available, an *impurity measure* is defined, which expresses the impurity as a real value. The decision trees algorithm we use (Witten and Frank, 2005), implements the C4.5 algorithm (Quinlan, 1993), which uses the entropy I as an impurity measure. The entropy of a set S with target class y which can take the values $y_1, \ldots, y_k$ is:

$$I(S) = \sum_{i=1}^{c} -\frac{p_i}{|S|} log_2 \frac{p_i}{|S|} \tag{C.1}$$

In this equation, p_i is the number of occurrences of y_i in S. Given this formula, the entropy gain is defined as the entropy of the original set minus the remaining entropy after splitting the set based on some attribute A:

$$gain(S, A) = I(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} I(S_v) \tag{C.2}$$

Here, S_v is the subset of S in which the value of attribute A is v in all examples. In the algorithm used, the attribute used to split a set is the one with the largest gain.

The second issue is when to stop splitting. If splitting continues until all leaf sets are pure, the decision tree will not likely generalize to unseen cases due to overfitting. One solution to this problem is stop splitting once the impurity of a leaf lies below a certain threshold. Another solution is to generate a very large tree, and prune branches that reflect noise or outliers. In this approach, a subset of the training examples is used to generate a very large decision tree, e.g. with pure sets at the leaves. Then, the remaining training data is used to prune the tree. Two leaf nodes are merged if the prediction error on the validation set is less with the resulting smaller tree than it was with the bigger tree (Quinlan, 1993).

For more information on decision trees, please see (Quinlan, 1993) or (Russell and Norvig, 2003). The WEKA implementation of the C4.5 algorithms we use is described in (Witten and Frank, 2005).

C.2 Regression and Model Trees

Regression trees may be considered as a variant of decision trees, designed to approximate real-valued functions instead of being used for classification tasks. Instead of a nominal value in each leaf, regression trees have a value which is the mean of the data examples in the partition. This representation requires a different splitting criterion. The algorithm chooses the split that partitions the data into two parts such that it minimizes the sum of the variance in the separate parts.

Model trees take it one step further, as their leaves represent line segments, representing the data in a partition (Quinlan, 1992). These line segments are acquired by performing standard multivariate linear regression on the examples in the partition. The impurity measure used to grow and prune model trees is:

Figure C.5. Model trees.

$$I(S) = \sum_{i:s_i \in S} (y_i - g(x_i))^2 \tag{C.3}$$

In which x_i are the attribute values the in example s_i, y_i the corresponding observed target value, and g is the value predicted by the line function. In principle g could be a more complex model, such as neural networks, but in practice this approach is seldom used (Belker, 2004).

C.3 Optimization of Model Trees

This section will describe an analytical procedure to find the minimum of a model tree, or sums of several model trees.

One way to determine the minimum of a model tree experimentally is to sample along all the dimensions (the variables with which it is called) in the model tree, and determine which combination of samples returns the lowest value. Of course, this minimum is only an approximation of the actual minimum. The higher the sampling rate, the higher its accuracy. Furthermore, sampling has a complexity of $O(n^d)$, in which n is the number of samples per dimension, and d the number of dimensions.

Our novel analytical method exploits the fact that a model tree is a set rules, each a bounded hyperplane. Determining the minimum of a bounded hyperplane is very easy: simply determine the values at the bounds, and take the minimum. Our approach is based on determining the minimum of each hyperplane, and then taking the minimum of all these values. This approach is $O(k)$, in which k is the number of hyperplanes, which is equivalent to the number of rules, or leaves in the model tree.

Figure C.6 shows a simple example for a one-dimensional search space, and three one-dimensional hyperplanes. In one-dimension, bounded hyperplanes are simply line segments.

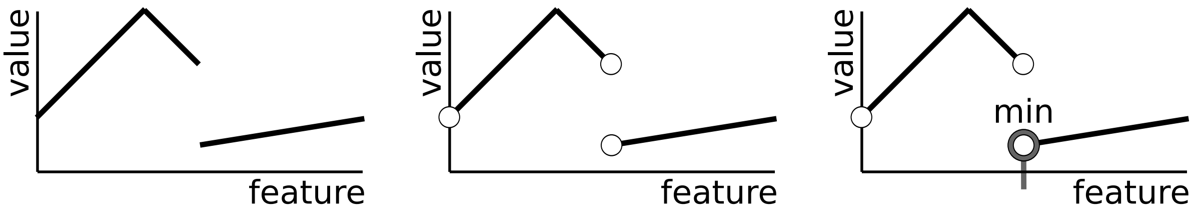


Figure C.6. Determining the minimum of a model tree. Instead of sampling along the x-axis, it is more efficient to determine the minimum of each line segment, and take the minimum of these minima.

Optimization of single Model Trees

In this section we will explain how this idea has been implemented. Below is fictional model tree, kept simple for reasons of clarity. Its format is the same as the resultbuffer in the WEKA program (Witten and Frank, 2005).

```
dist <= 1.52 :
|   dist <= 0.59 : 1.39*dist + 0.68*angle + 0.09
|   dist >  0.59 :
|   |   angle <= 0.62 : 1.35*dist + 0.22*angle + 0.13
|   |   angle >  0.62 : - 0.01*dist + 0.80*angle + 1.15
dist >  1.52 : 1.32*dist + 0.51*angle + 0.15
```

The first step is to convert the decision tree into a set of rules:

```
R1: (dist <= 1.52) & (dist <= 0.59) : 1.39*dist + 0.68*angle + 0.09
R2: (dist <= 1.52) & (dist > 0.59) & (angle <= 0.62) : 1.35*dist + 0.22*angle + 0.13
R3: (dist <= 1.52) & (dist > 0.59) & (angle > 0.62) : -0.01*dist + 0.80*angle + 1.15
R4: (dist > 1.52) : 1.32*dist + 0.51*angle + 0.15
```

Then, the minimum for each rule (hyperplane) is determined. We will use R3 as an example. First, we need to know the minimum and maximum values of all variables (e.g. $0 < dist < 3$, angle $0 < angle < PI$). In R3, the following ranges are valid.

```
R3: dist=[0.59..1.52], angle=[0.62..PI]
```

We determine the minimum of R3 by taking the extreme values in these ranges. The smallest value in the range is used if the variable is added, and the highest value if it is subtracted. This procedure is extremely fast, so there is little computation for each rule. For R3 the result is:

```
R3: -0.01*[0.59..1.52] + 0.80*[0.62..PI] + 1.15
     => -0.01*1.52 + 0.80*0.62 + 1.15 = 1.63
```

So, the minimum value R3 can reach is 1.63. For all the rules, these values are.

```
R1:  1.39*[0.00-0.59]+0.68*[0.00-  PI]+0.09  =>  1.39*0.00+0.68*0.00+0.09 = 0.09
R2:  1.35*[0.59-1.52]+0.22*[0.00-0.62]+0.13  =>  1.35*0.59+0.22*0.00+0.13 = 0.93
R3: -0.01*[0.59-1.52]+0.80*[0.62-  PI]+1.15  => -0.01*1.52+0.80*0.62+1.15 = 1.63
R4:  1.32*[1.52-3.00]+0.51*[0.00-  PI]+0.15  =>  1.32*1.52+0.51*0.00+0.15 = 2.16
```

The last step is to simply take the minimum of the rule minima (0.09, 0.93, 1.63, 2.16), which is 0.09. From R1, it can be read that this minimum is achieved with dist=0.00 and angle=0.00.

Processing bound variables

Often, some of the variables with which the model tree is called are already bound. For instance, the value of 'angle' might be 0.7. The procedure above does not change at all, since it operates on variable ranges, and the range of `angle` is simply defined to be [0.7,0.7]. As an added benefit, this knowledge makes computation faster, because we can eliminate all rules in which this value does not hold. In our simple example, angle=0.7 does not hold in R2.

```
R1: (dist <= 1.52) & (dist <= 0.59) : 1.39*dist + 0.68*angle + 0.09
R3: (dist <= 1.52) & (dist > 0.59) & (angle > 0.62) : -0.01*dist + 0.80*angle + 1.15
R4: (dist > 1.52) : 1.32*dist + 0.51*angle + 0.15
```

Then, as before, determine the ranges, choose the appropriate extreme value from this range, and voilá. Note that the angle has no real range, as it was set.

```
R1:  1.39*[0.00-0.59]+0.68*[0.70,0.70]+0.09 =>  1.39*0.00+0.68*0.70+0.09 = 0.57
R3: -0.01*[0.59-1.52]+0.80*[0.70,0.70]+1.15 => -0.01*1.52+0.80*0.70+1.15 = 1.69
R4:  1.32*[1.52-3.00]+0.51*[0.70,0.70]+0.15 =>  1.32*1.52+0.51*0.70+0.15 = 2.50
```

So, the minimum this model tree can have with an angle of 0.70 is 0.57, with dist=0.00.

Optimization of summations of Model Trees

In Section 5.3.2 on subgoal refinement, we saw that the minimum of the sum of two temporal prediction models of two consecutive actions was determined. This means that we need to determine the minimum of the sum of two model trees. This is done by first merging the two model trees into one, and then determining the minimum of the one model tree with the methods described above. The intuition behind this approach is shown in Figure C.7.

Instead of merging the model trees directly, they are first converted into sets of rules. These rulesets are then merged. Here is an example of two model trees, and their corresponding rulesets.

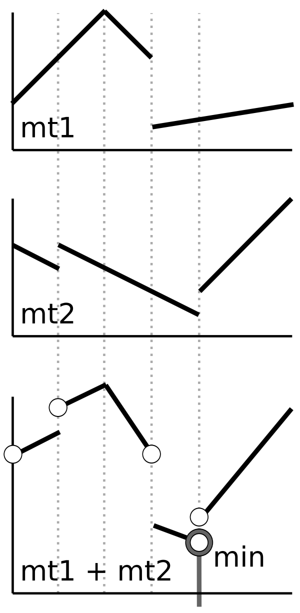


Figure C.7. Merging model trees

```
ModelTree 1                         RuleSet 1
a<=1 :
|   b<=3 : 3*a+2*b+1 (lm1)          (a<=1) & (b<=3) : 3*a+2*b+1
|   b>3 : 4*a+5*b+6 (lm2)    <=>    (a<=1) & (b>3) : 4*a+5*b+6
a>1 : 3*a+4*b+1 (lm3)               (a>1) : 3*a+4*b+1

ModelTree 2                         RuleSet 2
b<=2 : 1*a+1*b+1 (lm4)              (b<=2) : 1*a+1*b+1
b>2 :
|   a<=2 : 1*a+3*b+2 (lm5)   <=>    (b>2) & (a<=2) : 1*a+3*b+2
|   a>2 : 1*a+2*b+3 (lm6)           (b>2) & (a>2) : 1*a+2*b+3
```

Merging these two sets is done by first merging each rule of RuleSet1 with those of RuleSet2. The two lists of conditions are simply appended, and the linear models (`lm`) are summed. This yields the following set of rules:

```
                            RuleSet12 = RuleSet1 + RuleSet2
(a<=1) & (b<=3) : lm1
  (b<=2) : lm4              (a<=1) & (b<=3) & (b<=2) : lm1+lm4
  (b>2) & (a<=2) : lm5  =>  (a<=1) & (b<=3) & (b>2) & (a<=2) : lm1+lm5
  (b>2) & (a>2) : lm6       (a<=1) & (b<=3) & (b>2) & (a>2) : lm1+lm6

(a<=1) & (b>3) : lm2
  (b<=2) : lm4              (a<=1) & (b>3) & (b<=2) : lm2+lm4
  (b>2) & (a<=2) : lm5  =>  (a<=1) & (b>3) & (b>2) & (a<=2) : lm2+lm5
  (b>2) & (a>2) : lm6       (a<=1) & (b>3) & (b>2) & (a>2) : lm2+lm6

(a>1) : lm3
```

```
  (b<=2) : lm4              (a>1) & (b<=2) : lm3+lm4
  (b>2) & (a<=2) : lm5  =>  (a>1) & (b>2) & (a<=2) : lm3+lm5
  (b>2) & (a>2) : lm6       (a>1) & (b>2) & (a>2) : lm3+lm6
```

As can be seen, some of the lists of conditions contain contradictory conditions. For instance, in lm1+lm6, the conditions `(a<=1)` and `(a>2)` could never hold at the same time. Therefore, any new rule with such contradictions is removed (in this case lm1+lm6, lm2+lm4, lm2+lm6). This yields the six rules below. Summing the two linear models is easily done.

```
(a<=1) & (b<=2)   : lm1+lm4 = 4*a+3*b+2
(a<=1) & (2<b<=3) : lm1+lm5 = 4*a+5*b+3
(a<=1) & (b>3)    : lm2+lm5 = 5*a+8*b+8
(a>1) & (b<=2)    : lm3+lm4 = 4*a+5*b+2
(1<a<=2) & (b>2)  : lm3+lm5 = 4*a+7*b+3
(a>2) & (b>2)     : lm3+lm6 = 4*a+6*b+4
```

This procedure has been visualized in Figure C.8

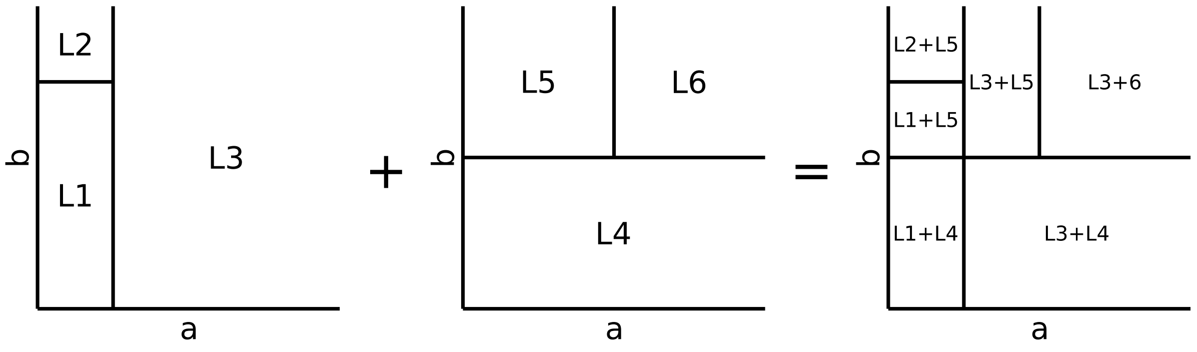


Figure C.8. Example of two merged model trees

The minimum of this ruleset can then be determined with the methods described in Section C.3. A downside of merging two rulesets is that the resulting ruleset will have many more rules. The worst case scenario is that merging two rulesets with $r1$ and $r2$ number of rules yield a ruleset with $r1 * r2$ rules. This happens for instance when the two rulesets have conditions on different variables. If conditions on variables are contradictory, rules can be eliminated, and the ruleset contains $< r1 * r2$ rules.

We have merged many temporal prediction models in the soccer domain, and the merged rulesets contain on average $0.4 * r1 * r2$ rules. The number of rules in the learned models trees is typically between 20 and 100, so merged rulesets contain between about 150 and 4000 rules. Note that determining the minimum of 4000 rules is usually much more efficient than optimizing in the variable space, especially for higher dimensions, as their complexities are $O(k)$ and $O(n^d)$ respectively.

D Summaries of Publications

We will now briefly present which systems, methods and results presented in this dissertation were published in which journals and conferences. The papers from 2004 are mostly on the enabling technologies. In 2005, the papers contain preliminary work and overviews. The final system and results described in this dissertation are presented in the papers from 2006 onwards.

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

An extensive journal article on the hardware, state estimation, and previous action selection module of the AGILO RoboCuppers. (Section 1.2.1)

(Utz et al., 2004) Utz, H., Stulp, F., and Mühlenfeld, A. (2004). Sharing belief in teams of heterogeneous robots. In Nardi, D., Riedmiller, M., and Sammut, C., editors, *RoboCup-2004: The Eighth RoboCup Competitions and Conferences*. Springer Verlag.

Description of belief state exchange requirements, and the implementation of the CORBA-based communication module. Joint publication with the University of Ulm and Universit of Graz. (Section 7.3.2)

(Stulp et al., 2004b) Stulp, F., Kirsch, A., Gedikli, S., and Beetz, M. (2004b). AGILO RoboCuppers 2004. In *RoboCup International Symposium 2004*, Lisbon.

Team Description Paper of the AGILO RoboCuppers at the RoboCup Competitions in Lisbon, Portugal. (Section 1.2.1)

(Stulp et al., 2004a) Stulp, F., Gedikli, S., and Beetz, M. (2004a). Evaluating multi-agent robotic systems using ground truth. In *Proceedings of the Workshop on Methods and Technology for Empirical Evaluation of Multi-agent Systems and Multi-robot Teams (MTEE)*.

Implementation and evaluation of the ground truth system. (Section B.3)

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

First subgoal refinement results in the simulated soccer domain. (Chapter 5)

(Stulp and Beetz, 2005c) Stulp, F. and Beetz, M. (2005c). Tailoring action parameterizations to their task contexts. IJCAI Workshop "Agents in Real-Time and Dynamic Environments".

An overview of all applications of action models in one coherent computational model. (Overview of Dissertation)

(Stulp and Beetz, 2005a) Stulp, F. and Beetz, M. (2005a). Optimized execution of action chains through subgoal refinement. ICAPS Workshop "Plan Execution: A Reality Check".

A brief overview of subgoal refinement from a planning perspective. (Chapter 5)

(Stulp and Beetz, 2006) Stulp, F. and Beetz, M. (2006). Action awareness – enabling agents to optimize, transform, and coordinate plans. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

A brief overview of subgoal refinement and implicit coordination. (Overview of Dissertation)

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

Extensive evaluation of implicit coordination within the AGILO RoboCuppers. (Chapter 7)

(Isik et al., 2006) Isik, M., Stulp, F., Mayer, G., and Utz, H. (2006). Coordination without negotiation in teams of heterogeneous robots. In *Proceedings of the RoboCup Symposium*.

Integrates and extends the results of (Utz et al., 2004) and (Stulp et al., 2006a) by evaluating implicit coordination in heterogeneous teams. (Chapter 7)

(Stulp et al., 2006b) Stulp, F., Pflüger, M., and Beetz, M. (2006b). Feature space generation using equation discovery. In *Proceedings of the 29th German Conference on Artificial Intelligence (KI)*.

Implementation and evaluation of the directed equation discovery system for generating appropriate feature spaces. (Section 4.1.1)

(Stulp et al., 2007) Stulp, F., Koska, W., Maldonado, A., and Beetz, M. (2007). Seamless execution of action sequences. In *Accepted for the IEEE International Conference on Robotics and Automation (ICRA)*. to appear.

Subgoal refinement integrated in the PDDL planner VHPOP. First results on real soccer robots. Further evaluation in the service robotics and arm control domain. (Section 5.2.1)

Bibliography

ActivMedia Robotics (1998). Pioneer 1/at operations manual, ed. 2. `http://robots.mobilerobots.com/docs/all_docs/opmanv2.pdf`.

Amtec Robotics (2005). Programmers guide for PowerCube. `http://www.amtec-robotics.com/files/Programmers guide for PowerCube.pdf`.

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

Arai, T. and Stolzenburg, F. (2002). Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In C. Castelfranchi, W. L. J., editor, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 11–18.

Ariff, G., Donchin, O., Nanayakkara, T., and Shadmehr, R. (2002). A real-time state predictor in motor control: study of saccadic eye movements during unseen reaching movements. *Journal of Neuroscience*, 22:7721–7729.

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

Auster, P. (1987). *The New York Trilogy*. Faber & Faber.

Baerends, G. P. (1970). A model of the functional organization of incubation behaviour. *Behaviour Supplement*, 17:263–312.

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

Balac, N., Gaines, D., and Fisher, D. (2000). Learning action models for navigation in noisy environments. In *In ICML Workshop on Machine Learning of Spatial Knowledge*.

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

Beetz, M. (1999). Structured Reactive Controllers — a computational model of everyday activity. In Etzioni, O., Müller, J., and Bradshaw, J., editors, *Proceedings of the Third International Conference on Autonomous Agents*, pages 228–235.

Beetz, M. (2000). *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume 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. and 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., Gedikli, S., Hanek, R., Schmitt, T., and Stulp, F. (2003a). AGILO RoboCuppers 2003: Computational Priciples and Research Directions. In *RoboCup International Symposium 2003*, Padova.

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

Beetz, M., Stulp, F., Kirsch, A., Müller, A., and Buck, S. (2003b). Autonomous robot controllers capable of acquiring repertoires of complex skills. In *RoboCup International Symposium 2003*, Padova.

Behnke, S. and Rojas, R. (2001). A hierarchy of reactive behaviors handles complexity. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems, From RoboCup to Real-World Applications (selected papers from the ECAI 2000 Workshop and additional contributions)*.

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

Belker, T., Hammel, M., and Hertzberg, J. (2003). Learning to optimize mobile robot navigation based on htn plans. In *Proceedings of the International Conference on Robotics and Automation (ICRA03)*.

Bell, C. C., Han, V. Z., Sugawara, Y., and Grant, K. (1997). Synaptic plasticity in a cerebellum-like structure depends on temporal order. *Nature*, 387:278?281.

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

Bloedorn, E. and Michalski, R. S. (1998). Data-driven constructive induction. *IEEE Intelligent Systems*, 13(2):30–37.

Blumberg, B. M. (2003). *D-Learning: what learning in dogs tells us about building characters that learn what they ought to learn*, pages 37–67. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Botelho, S. C. and Alami, R. (1999). M+: A scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA99)*.

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

Brachman, R. (2002). Systems that know what they're doing. *IEEE Intelligent Systems*, pages 67 – 71.

Brock, O. and Khatib, O. (1999). Elastic Strips: A framework for integrated planning and execution. In *Proceedings International Symposium on Experimental Robotics*.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23.

Bruegge, B. and Dutoit, A. H. (2003). *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall.

Buck, S. (2003). *Experience-Based Control and Coordination of Autonomous Mobile Systems in Dynamic Environments*. PhD thesis, Department of Informatics, Technische Universität München.

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

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

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

Buck, S., Stulp, F., Beetz, M., and Schmitt, T. (2002c). Machine Control Using Radial Basis Value Functions and Inverse State Projection. In *Proc. of the IEEE Intl. Conf. on Automation, Robotics, Control, and Vision*.

Cambon, S., Gravot, F., and Alami, R. (2004). A robot task planner that merges symbolic and geometric reasoning. In *ECAI*, pages 895–899.

Carpenter, P., Riley, P. F., Veloso, M., and Kaminka, G. (2002). Integration of advice in an action-selection architecture. In *Proceedings of the 2002 RoboCup Symposium*.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Castelpietra, C., Iocchi, L., Nardi, D., Piaggio, M., Scalzo, A., and Sgorbissa, A. (2000). Coordination among heterogenous robotic soccer players. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2000)*.

Cavaco, S., Anderson, S., Allen, J., Castro-Caldas, A., and Damasio, H. (2004). The scope of preserved procedural memory in amnesia. *Brain*, 127:1853–67.

Chaimowicz, L., Campos, M. F. M., and Kumar, V. (2002). Dynamic role assignment for cooperative robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA02)*.

Choi, W. and Latombe, J. (1991). A reactive architecture for planning and executing robot motion with incomplete knowledge. In *In IEEE/RSJ International Workshop on Intelligent Robots and Systems*, pages 24–29.

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

Cortés, U., Annicchiarico, R., Vázquez-Salceda, J., Urdiales, C., namero, L. C., López, M., Sànchez-Marrè, M., and Caltagirone, C. (2003). Assistive technologies for the disabled and for the new generation of senior citizens: the e-tools architecture. *AiCommunications*, 16(3):193–207.

Cosy (2004). Cosy homepage. `www.cognitivesystems.org`.

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

Dearden, A. (2006). Personal communication.

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

Dennett, D. (1987). *The Intentional Stance*. MIT-Press.

Dias, M. B. and Stentz, A. (2001). A market approach to multirobot coordination. Technical Report CMU-RI -TR-01-26, Robotics Institute, Carnegie Mellon University.

Dietel, M., Gutmann, S., and Nebel, B. (2002). CS Freiburg: Global view by cooperative sensing. In *Proceedings of RoboCup International Symposium 2001*, volume 2377 of *LNAI*, pages 133–143. Springer-Verlag.

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

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

F180 Laws (2004). Laws of the RoboCup F180 league 2004. http://www.itee.uq.edu.au/˜wyeth/F180

Fikes, R. O. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Technical Report 43r, AI Center, SRI International.

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

Flanagan, J. R., Vetter, P., Johansson, R. S., and Wolpert, D. M. (2003). Prediction precedes control in motor learning. *Current Biology*, 13:146–150.

Flash, T. and Hogan, N. (1985). The coordination of arm movements - an experimentally confirmed mathematical model. *Journal of Neuroscience*, 5:1688–1703.

Fong, T., Nourbakhsh, I., Kunz, C., Flückiger, L., and Schreiner, J. (2005). The peer-to-peer human-robot interaction project. *Space*.

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

Fox, M., Gough, J., and Long, D. (2006b). Using learned action models in execution monitoring. In *Proceedings of UK Planning and Scheduling SIG*.

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

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

Gabrieli, J. D. E., Corkin, S., Mickel, S. F., , and Growdon, J. H. (2004). Intact acquisition and long-term retention of mirror-tracing skill in alzheimer's disease and in global amnesia. *Behavioral Neuroscience*, 107(6):899–910.

Gallego, J. and Perruchet, P. (2006). Do dogs know related rates rather than optimization? *College Mathematics Journal*.

Gerkey, B., Vaughan, R. T., and 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 (ICAR2003)*, pages 317–323.

Gerkey, B. P. and Matarić, M. J. (2003). Multi-robot task allocation: Analyzing the complexity and optimality of key architectures. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

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

Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11(1):23–63.

Großmann, A. (2001). *Continual learning for mobile robots*. PhD thesis, School of Computer Science, The University of Birmingham.

Grounds, M. and Kudenko, D. (2005). Combining reinforcement learning with symbolic planning. In *Proceedings of the Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*.

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

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

Haruno, M., Wolpert, D. M., and 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*, pages 31–37, Cambridge, MA, USA. MIT Press.

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

Hawkins, J. and Blakeslee, S. (2004). *On Intelligence*. Times Books.

Helmholtz, H. v. (1896). *Handbuch der physiologischen optik*. L. Voss.

Hoffmann, J. (2003). The metric-FF planning system: Translating ignoring delete lists to numerical state variables. *Journal of Artificial Intelligence Research*, 20.

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

Hooper, R. (1994). *Multicriteria Inverse Kinematics for General Serial Robots*. PhD thesis, University of Texas.

Infantes, G., Ingrand, F., and Ghallab, M. (2006). Learning behaviors models for robot execution control. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*.

Isik, M. (2005). Installation eines CORBA basierten frameworks (MIRO) auf den Agilo robocup robotern. Software Development Project (SEP) Report, Technische Universiät München.

Isik, M. (2006). Implizite koordination innerhalb heterogener roboterteams. Master's thesis, Technische Universiät München.

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

Jacobs, R. A. and 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. and Christaller, T. (1998). Dual dynamics: Designing behavior systems for autonomous robots. *Artificial Life and Robotics*.

John, G. H., Kohavi, R., and Pfleger, K. (1994). Irrelevant features and the subset selection problem. In *Proceedings of International Conference on Machine Learning*, pages 121–129.

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

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45.

Kambhampati, S., Knoblock, C. A., and Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76(1–2):167–238.

Kaplan, F., P-Y., O., Revel, A., Gaussier, P., Nadel, J., Berthouze, L., Kozima, H., Prince, G. C., and Balkenius, C., editors (2006). *Proceedings of the Sixth International Conference on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*. Lund University Cognitive Studies.

Karlsson, L. and Schiavinotto, T. (2002). Progressive planning for mobile robots: a progress report.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The robot world cup initiative. In *Agents*, pages 340–347.

Kleiner, A., Dietl, M., and Nebel, B. (2002). Towards a life-long learning soccer agent. In *Proceedings of the International RoboCup Symposium 2002*.

Kobialka, H.-U. and Jaeger, H. (2003). Experiences using the dynamical system paradigm for programming RoboCup robots. In *Proceedings of the 2nd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE)*, pages 193–202.

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

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

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

Kraetzschmar, G. K., Mayer, G., Utz, H., Baer, P., Clauss, M., Kaufmann, U., Lauer, M., Natterer, S., Przewoznik, S., Reichle, R., Sitter, C., Sterk, F., and Palm, G. (2004). The Ulm Sparrows 2004 - team description paper. In Nardi, D., Riedmiller, M., Sammut, C., and Santos-Victor, J., editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, Germany. Springer-Verlag.

Kushmerick, N., Hanks, S., and Weld, D. (1994). An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1073–1078, Seattle, Washington, USA. AAAI Press/MIT Press.

Laird, J., Rosenbloom, P., and Newell, A. (1986). Chunking in SOAR: the anatomy of a general learning mechanism. *Machine Learning*, 1:11–46.

Langley, P., Simon, H., Bradshaw, G., and Zytkow, J. (1987). *Scientific Discovery: Computational Explorations of the Creative Processes*. MIT Press.

Liu, H. and Motoda, H. (1998). Feature transformation and subset selection. *IEEE Intelligent Systems*, 13(2):26–28.

Lopes, M. and Santos-Victor, J. (2005). Visual learning by imitation with motor representations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 35(3):438–449.

Lötzsch, M., Bach, J., Burkhard, H.-D., and Jüngel, M. (2004). Designing agent behavior with the extensible agent behavior specification language XABSL. In *7th International Workshop on RoboCup 2003*.

Metta, G., Sandini, G., Vernon, D., Caldwell, D., Tsagarakis, N., Beira, R., Santos-Victor, J., Ijspeert, A., Righetti, L., Cappiello, G., Stellin, G., and Becchi, F. (2006). The RobotCub project – an open framework for research in embodied cognition. In *Humanoids Workshop, Proceedings of the IEEE–RAS International Conference on Humanoid Robots*.

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

Murray, J. (2001). Specifying agents with UML in robotic soccer. Technical Report 10-2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz.

Murray, J. and Stolzenburg, F. (2005). Hybrid state machines with timed synchronization for multi-robot system specification. In Reis, L. P., Carreto, C., Silva, E., and Lau, N., editors, *Proceedings of Workshop on Intelligent Robotics (IROBOT2005)*.

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

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

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

Nourbakhsh, I., Szcara, K., Koes, M., Yong, M., Lewis, M., and Burion, S. (2005). Human-robot teaming for search and rescue. *Pervasive Computing*.

Obst, O. (2002). Specifying rational agents with statecharts and utility functions. In A. Birk, S. Coradeschi, S. T., editor, *RoboCup 2001 Robot Soccer World Cup V*, pages 173?–182.

Parker, L. E. (1998). Alliance: An architecture for fault-tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240.

Parr, R. (1998). *Hierarchical Control and learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley.

Pennings, T. (2007). Personal communication.

Pennings, T. J. (2003). Do dogs know calculus? *College Mathematics Journal*, 4(May):178–182.

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

Pflüger, M. (2006). Feature space transformation using directed equation discovery. Master's thesis, Technische Universiät München.

Pratt, L. Y. and Jennings, B. (1996). A survey of transfer between connectionist networks. *Connection Science*, 8(2):163–184.

Qin, S. J. and Badgwell, T. J. (1998). An overview of nonlinear model predictive control applications. Presented at Nonlinear MPC Workshop.

Quinlan, R. (1992). Learning with continuous classes. In Adams, A. and Sterling, L., editors, *Proceedings of the* 5^{th} *Australian Joint Conference on Artificial Intelligence*, pages 343–348.

Quinlan, R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California.

Rao, R., Shon, A., and Meltzoff, A. (2005). A bayesian model of imitation in infants and robots. In *Imitation and Social Learning in Robots, Humans, and Animals*. Cambridge University Press.

Roboteq Inc. (2004). AX2550 datasheet. `http://www.roboteq.com/files/datasheets/ax2550datashe`

Röfer, T. (2002). An architecture for a national RoboCup team. In *Proceedings of RoboCup International Symposium 2002*, volume 2752 of *LNAI*, pages 417–425. Springer-Verlag.

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

Rusu, R. B. (2006). Acquiring models of everyday activities for robotic control in 'current PhD research in pervasive computing'. *Technical Reports - University of Munich, Department of Computer Science, Media Informatics Group*, LMU-MI-2005-3.

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

Ryan, M. R. K. (2004). *Hierarchical Reinforcement Learning: A Hybrid Approach*. PhD thesis, University of New South Wales, School of Computer Science and Engineering.

Ryan, M. R. K. and Reid, M. D. (2000). Learning to fly: An application of hierarchical reinforcement learning. In *Proceedings of the 17th International Conference of Machine Learning*.

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

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

Schmidt, D. C., Gokhale, A., Harrison, T., and Parulkar, G. (1997). A high-performance endsystem architecture for real-time CORBA. *IEEE Comm. Magazine*, 14(2).

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

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

Scholz, J. and Schöner, G. (1999). The uncontrolled manifold concept: identifying control variables for a functional task. *Exp Brain Res*, 126(3):289–306.

Scoville, W. B. and Milner, B. (1957). Loss of recent memory after bilateral hippocampal lesions. *Journal of Neurology, Neurosurgery and Psychiatry*, 20:11–21.

Sen, S., Sekaran, M., and Hale, J. (1994). Learning to coordinate without sharing information. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 426–431.

Servan-Schreiber, E. and Anderson, J. R. (1990). Chunking as a mechanism of implicit learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 16:592–608.

Shahaf, D. and Amir, E. (2006). Learning partially observable action schemas. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*.

Shapiro, A., Pighin, F., , and Faloutsos, P. (2003). Hybrid control for interactive character animation. In *Pacific Graphics*, pages 455–461.

Shen, W.-M. (1994). *Autonomous Learning from the Environment*. W. H. Freeman & Co., New York, NY, USA.

Silver, D. L. and Mercer, R. E. (1998). The task rehearsal method of sequential learning.

Siméon, T., Laumond, J.-P., and Lamiraux, F. (2001). Move3D: a generic platform for path planning. In *4th International Symposium on Assembly and Task Planning*.

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

Sloman, A. (2006). Architecture of brain and mind integrating high level cognitive processes with brain mechanisms and functions in a working robot. Technical Report COSY-TR-0602, University of Birmingham.

Smith, D., Frank, J., and Jónsson, A. (2000). Bridging the gap between planning and scheduling. *Knowledge Engineering Review*.

Smith, R. (2004). Open dynamics engine. `http://www.ode.org`.

Spaan, M. T. J. and Groen, F. C. A. (2002). Team coordination among robotic soccer players. In Kaminka, G., Lima, P. U., and Rojas, R., editors, *Proceedings of RoboCup International Symposium 2002*.

Stone, P. and Veloso, M. (1999). Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273.

Stulp, F. and Beetz, M. (2005a). Optimized execution of action chains through subgoal refinement. ICAPS Workshop “Plan Execution: A Reality Check”.

Stulp, F. and Beetz, M. (2005b). 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. and Beetz, M. (2005c). Tailoring action parameterizations to their task contexts. IJCAI Workshop “Agents in Real-Time and Dynamic Environments”.

Stulp, F. and Beetz, M. (2006). Action awareness – enabling agents to optimize, transform, and coordinate plans. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Stulp, F., Gedikli, S., and Beetz, M. (2004a). Evaluating multi-agent robotic systems using ground truth. In *Proceedings of the Workshop on Methods and Technology for Empirical Evaluation of Multi-agent Systems and Multi-robot Teams (MTEE)*.

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

Stulp, F., Kirsch, A., Gedikli, S., and Beetz, M. (2004b). AGILO RoboCuppers 2004. In *RoboCup International Symposium 2004,* Lisbon.

Stulp, F., Koska, W., Maldonado, A., and Beetz, M. (2007). Seamless execution of action sequences. In *Accepted for the IEEE International Conference on Robotics and Automation (ICRA)*. to appear.

Stulp, F., Pflüger, M., and Beetz, M. (2006b). Feature space generation using equation discovery. In *Proceedings of the 29th German Conference on Artificial Intelligence (KI)*.

Stulp, F., Utz, H., and Nebel, B., editors (2004c). *Proceedings of the Workshop on Methods and Technology for Empirical Evaluation of Multi-agent Systems and Multi-robot Teams (MTEE)*. Springer. In conjunction with the 27th German Conference on Artificial Intelligence.

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

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: an Introduction*. MIT Press.

Sutton, R. S., Precup, D., and Singh, S. P. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.

Tews, A. and Wyeth, G. (2000). Thinking as one: Coordination of multiple mobile robots by shared representations. In *Intl. Conf. on Robotics and Systems (IROS)*.

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

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., and Mahoney, P. (2006). Stanley, the robot that won the DARPA grand challenge. *Journal of Field Robotics*.

Trafton, J. G., Cassimatis, N. L., Bugasjska, M. D., Brock, D. P., Mintz, F. E., and Schultz, A. C. (2005). Enabling effective human-robot interaction using perspective taking in robots. *IEEE Transactions on Systems, Man and Cybernetics*, 35(4):460–470.

Trolltech (2005). Qt homepage. `www.trolltech.com/products/qt`.

Uno, Y., Wolpert, D. M., Kawato, M., and 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., abd Gerd Mayer, G. K., and Palm, G. (2005). Hierarchical behavior organization. In *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS)*.

Utz, H., Sablatnög, S., Enderle, S., and Kraetzschmar, G. K. (2002). Miro – middleware for mobile robot applications. *IEEE Trans. on Robotics and Automation*, 18:493–497.

Utz, H., Stulp, F., and Mühlenfeld, A. (2004). Sharing belief in teams of heterogeneous robots. In Nardi, D., Riedmiller, M., and Sammut, C., editors, *RoboCup-2004: The Eighth RoboCup Competitions and Conferences*. Springer Verlag.

Vail, D. and Veloso, M. (2003). Multi-robot dynamic role assignment and coordination through shared potential fields. In *Multi-Robot Systems*. Kluwer.

Veloso, M., Stone, P., and Bowlin, M. (1999). Anticipation as a key for collaboration in a team of agents: A case study in robotic soccer. In *Proceedings of SPIE Sensor Fusion and Decentralized Control in Robotic Systems II*.

Vilalta, R. and Drissi, Y. (2002). A perspective view and survey of metalearning. *Artificial Intelligence Review*.

Weld, D. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.

Werger, B. B. and Matarić, M. J. (2000). Broadcast of local eligibility: behavior-based control for strongly cooperative robot teams. In *AGENTS ’00: Proceedings of the fourth international conference on Autonomous agents*, pages 21–22.

Winograd, T. (1975). Frame representations and the procedural/declarative controversy. In Bobrow, D. G. and Collins, A., editors, *Representation and Understanding: Studies in Cognitive Science*, pages 185–210. Academic Press, New York.

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

Wolpert, D. and Ghahramani, Z. (2000). Computational principles of movement neuroscience. *Nature Neuroscience Supplement*, 3.

Wolpert, D. M., Doya, K., and Kawato, M. (2003). A unifying computational framework for motor control and social interaction. *Philosophical Transactions of the Royal Society*, 358:593–602.

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

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