Robot Controllers for Highly Dynamic Environments With Real-time Constraints

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Alexander Antoine Ferrein

aus Hilden

Berichter: Universitätsprofessor Gerhard Lakemeyer, Ph.D.
Universitätsprofessor Michael Beetz, Ph.D.

Tag der mündlichen Prüfung: 20. Dezember 2007

TO MY FAMILY

Acknowledgements

First of all, I would like to thank my supervisor Gerhard Lakemeyer for several reasons. He convinced me to start the quest on mobile robots and with this giving me this great opportunity and experience of being part of the RoboCup project. During the recent years he always gave the right advice at the right time in his very subtle way. His way of giving sublte (mostly) reinforcement (positive as well as negative) very much enforced my scientific education and development, and I would like to thank him for the trust that our efforts will come to some good end, and for his patience and understanding that sometimes making things run in reality takes longer than expected. In this context I have to underline that I am very grateful for his respect of the practical work we have done during the last years. His passion to sit in early morning hours with the RoboCup team to debug something at some tournament site is well remembered. Finally, I would like to thank him for his valuable comments on earlier versions of this thesis, helping to make it a coherent and hopefully (formally) sound document. I am grateful to Michael Beetz who kindly accepted the task of the second reviewer of this thesis.

I'm indebted to many people supporting this work during the last years. It may sound a little bit like an acceptance speech at the Grammys, but I *do* owe a lot to the following people. I have to thank the Knowledge-based Systems Group for the great support of our endevours, in particular (in the order of appearance) Gero Iwan, Hendrik Grosskreutz, Günter Gans, Frank Dylla, Christian Fritz, Vazha Amiranashvili, Jens Classen, Stefan Schiffer, and Daniel Beck; Dominik Schmitz for listening to my complaints on many train rides, for his valuable comments and solution ideas on several problems; and the whole group of people working at Computer Science V. I have to thank all the members of the AllemaniACs RoboCup team. The members over the last five years were (in alphabetical order): Daniel Beck, Florin Bley, Masrur Doostdar, Frank Dylla, Benjamin von Eicken, Christian Fritz, Carsten Gester, Claudia Gönner, Lutz Hermanns, Yuxio Hu, Stefan Jacobs, Mirko Kohns, Savas Konur, Alexandros Matsikis, Christoph Mies, Tim Niemüller, Martin Rous, Stefan Schiffer, Nils Springob, Andreas Strack, Pei Tang, Philipp Vorst, and Jost Wunderlich. With such a long list of people one cannot pinpoint the reasons for my gratefulness in detail. I just want to remark that some of these people I owe much and that these people should know that I know! Thanks for the good times, but also for the hard times we went through together. I was able to learn a lot from all of them in so many respects; the working together (which was not always a pleasure) let me learn so much and will be well remembered. I would also like to thank the AllemaniACs@Home team (Stefan and Tim and Masrur) explicitly for their outstanding performance during last RoboCups showing that our robots are not only bulky but that the approach works in practice!

Next, I want to thank the German Science Foundation (DFG) and the Ministry of Science and Education (MSWF) of North-Rhine Wesfalia for the financial support of the RoboCup project. Further, I'm also grateful for the financial support of Computer Science V, personified through Mathias Jarke and Gerhard Lakemeyer, and the Bonn-Aachen Institute of Technology.

Thanks to Tom Waits and to Linkin Park for providing the right background atmosphere during the long hours of the writing this document.

I am deeply indebted to Tessa Oliver inter alia for proof-reading the whole thesis. Her comments and advice finally made this document 'human-readable'. I want to thank my family for their support during recent years. In the end, I have to thank Tanja for her love and her support especially during the time of writing this thesis, always having understanding for “I'd like to but, ...”.

Abstract

The fields of mobile autonomous robotics and cognitive robotics are active research fields. In the last decade progress has become more and more visible. The reason for this is that the systems became more affordable, there were side effects to automobile industries (driving assistance systems) which make use of similar methods, and that activities like RoboCup (robotic soccer) became more present in the media. Additionally, there are efforts to standardize software and hardware components. In recent years several successful applications showed that mobile robots can interact with their environment and fulfill meaningful and useful tasks. Nevertheless, many questions on how to design autonomous mobile robots remain open and are subject to active research in this field. For problems like navigation, collision avoidance, and localization, robust approaches have been proposed which are widely used. The question of how such robots can act intelligently, has wide-spread ideas and approaches.

In this thesis we propose an approach to the problem of intelligent decision making (deliberation) for robots or agents which moreover have to decide under real-time constraints in adversarial domains, i.e. multi-agent domains where opponents have contrary goals and try to foil the goals of the opposing team. Agents can be seen as non-embodied robots acting in simulated environments. Our account is based on the logical agent high-level programming language Golog. Golog is based on the situation calculus, a powerful logical calculus for reasoning about actions and change. During recent years several extensions for integrating concurrency and sensing, dealing with continuous change, and applying decision theory have been made. The question we are concerned with in this thesis is how these different approaches can be put together in such a way that an agent is enabled to come to intelligent decisions while acting in dynamic, adversarial real-time domains. The crucial point is that deliberation in general is computationally expensive, but nevertheless is needed to come to intelligent decisions. On the other hand, the robot must be able to react quickly to changes in the environment, because otherwise an opponent might take advantage.

We propose the language READYLOG as an approach to intelligent decision making especially in dynamic real-time domains. READYLOG is a GOLOG family language and combines features known from other dialects in one framework. It is based on the situation calculus and offers probabilistically projecting a given world situation into the future, dealing with a continuously changing world, coming with an efficient implementation and a mechanism to progress the internal database. READYLOG especially makes use of decision-theoretic planning as a means to intelligent decision making. Several alternatives in the robot program are left open and READYLOG chooses the most promising one against a background optimization theory. As decision-theoretic planning is computationally costly, we are looking for ways to reduce the complexity of planning. One possibility which we propose is to make use of options, which are macro-actions in the decision-theoretic context.

We show several detailed applications of READYLOG in dynamic real-time domains. These range from interactive computer games to robotic soccer, simulated as well as with real robots. READYLOG has been successfully applied for several soccer agents which participated in the international RoboCup world championships. As we are also dealing with real robots, we also discuss several new approaches and extensions in the field of robotics, like a new robust laser-based navigation algorithm. Finally, as READYLOG comes with a formal logical semantics it is well-suited to model behavior in a general way. We started to develop a formal soccer theory for robots based on human soccer knowledge. The formalization language we used is READYLOG.

Zusammenfassung

Die Forschungsfelder „Mobile Autonome Robotik“ und „Kognitive Robotik“ sind aktive Felder und werden intensiv bearbeitet. Der Fortschritt auf diesen Gebieten ist besonders in den letzten 10 Jahren sichtbar geworden. Ein Grund für die verstärkte Wahrnehmbarkeit dieser Forschungsgebiete ist, dass Robotersysteme günstiger in der Anschaffung und Entwicklung werden. Es gibt vermehrt Querbeziehungen zur Automobilindustrie im Bereich von Fahrerassistenzsystemen, wo ähnliche Methoden für ähnliche Problemstellungen eingesetzt werden. Davon profitiert die Roboterforschung u.a., da durch die Massenproduktion in der Automobilindustrie z.B. Sensorik billiger wird. Weiterhin sind Initiativen wie *RoboCup*, die Weltmeisterschaft der Fußballroboter, stärker präsent in den Medien. Darüber hinaus gibt es Bemühungen, sowohl Robotik-Software als auch Robotik-Hardware zu standardisieren und damit für eine breitere Basis verfügbar zu machen. Anwendungen zeigen, dass mobile Roboter in der Lage sind, mit ihrer Umwelt zu interagieren und Aufgaben erfolgreich zu erfüllen. Eine Vielzahl offener Probleme existiert, und nach Lösungen wird intensiv geforscht. Für Probleme wie Navigation, Kollisionsvermeidung oder Lokalisierung gibt es mittlerweile robuste Algorithmen. Welche Methoden Agenten oder Roboter intelligent handeln lassen, ist auf der anderen Seite noch nicht so offensichtlich.

In dieser Arbeit wird ein Ansatz zur intelligenten Entscheidungsfindung (Deliberation) für Roboter oder Softwareagenten, die Entscheidungen unter Echtzeitbedingungen unter Einfluss von Gegenspielern treffen müssen, vorgeschlagen. Agenten sind hier als nicht-physikalische Roboter, die in simulierten Umgebungen agieren, zu sehen. Der vorgestellte Ansatz fußt auf der logikbasierten Hochsprache GOLOG. GOLOG selber basiert auf dem Situationenkalkül, ein mächtiges logisches Kalkül, um Schlüsse über Aktionen und deren Effekte zu ziehen. Für GOLOG wurden verschiedene Erweiterungen vorgeschlagen, die kontinuierliche Domänen betrachten, nebenläufiges Ausführen von Aktionen erlauben oder entscheidungstheoretisches Planen unterstützen. Eine Frage, der in dieser Arbeit nachgegangen wird, ist, wie man diese unterschiedlichen Erweiterungen integrieren kann, so dass ein Agent, der in einer Echtzeitdomäne, in der auch Gegenspieler agieren, zu intelligenten Entscheidungen kommen kann. Das Hauptproblem liegt darin, dass Deliberation im Allgemeinen rechenintensiv ist. Sie wird jedoch benötigt, um zu intelligenten Entscheidungen zu kommen. Auf der anderen Seite muss ein Agent oder Roboter in solchen Domänen schnell reagieren, da der Gegner sonst jede Verzögerung ausnutzen könnte.

Wir schlagen mit dieser Arbeit die Sprache READYLOG als einen Ansatz zur intelligenten Entscheidungsfindung in dynamischen Echtzeitdomänen vor. READYLOG gehört der GOLOG-Sprachfamilie an und vereint verschiedene Charakteristiken, die von anderen Dialekten bekannt sind. Wie GOLOG beruht READYLOG auf dem Situationenkalkül und bietet die Möglichkeit, probabilistische Projektionen durchzuführen oder auf eine sich kontinuierlich verändernde Welt zu reagieren. Das Laufzeitsystem von READYLOG implementiert darüber hinaus einen Progressionsmechanismus der internen Wissensbasis. Eines der Hauptmerkmale von READYLOG ist die Möglichkeit zum entscheidungstheoretischen Planen. Das Agentenprogramm lässt verschiedene Handlungsalternativen offen. Die Beste dieser Alternativen wird mittels einer Optimierungstheorie von READYLOG zur Laufzeit ausgewählt. Da entscheidungstheoretisches Planen rechenintensiv ist, suchen wir weiterhin nach Wegen, die Komplexität des Planens zu senken. Eine Möglichkeit, von der Gebrauch gemacht wird, ist die Einführung von sogenannte Makroaktionen oder Optionen.

Im weiteren Verlauf der Arbeit werden in verschiedenen detailliert beschriebenen Anwendungen die Möglichkeiten des Einsatzes von READYLOG in dynamischen Echtzeitdomänen aufgezeigt. Die Anwendungen reichen von Agenten in interaktiven Computerspielen bis hin zu Roboterfußball, sowohl mit Softwareagenten als auch mit ‚echten' Robotern. READYLOG wurde erfolgreich zur Implementierung von Fußballrobotern, die bei internationalen RoboCup-Meisterschaften teilnehmen, genutzt. Da sich unsere Anwendungen auf Roboter beziehen und einige Beiträge auf diesem Gebiet u.a. zur laser-basierten Navigation gemacht wurden, befasst sich ein Teil der Arbeit mit mobilen Robotern. Ein anderer Teil beschreibt den Einsatz von READYLOG zur Verhaltensspezifikation. Wir haben READYLOG mit der zugrundeliegenden formalen Semantik dazu benutzt, Fußballstrategien zu formalisieren, um diese dann auf dem Roboter einsetzen zu können und dadurch das Verhalten der Roboter zu verbessern. READYLOG erwies sich hier als sehr ausdrucksstark.

Contents

List of Figures

List of Tables

QUOD QUALE SIT, NON EST MEUM DICERE, PROPTEREA QUOD MINIME SIBI QUISQUE NOTUS EST ET DIFFICILLIME DE SE QUISQUE SENTIT.

Marcus Tullius Cicero, *De Oratore,* III, 33

What the difference might be is not for me to say, for the reason that everyone knows themselves the least and it is the hardest to judge oneself.

Worin der Unterschied bestehe, kommt mir nicht zu, zu sagen, deshalb, weil jeder sich am wenigsten kennt und am schwierigsten über sich selbst urteilt.

Chapter 1

Introduction

Mobile robotics has been an active research field for the last four decades. One major application for robots is in production plants. Robots solder and braze cars with high precision and also varnish components. These robots raise the efficiency of automation. Besides possible negative effects for the employment markets of current economies, it has the positive effect that robots can fulfill tasks which are too dangerous or unsanitary for humans. The more interesting field, from the Artificial Intelligence (AI) research perspective, is the field of mobile autonomous intelligent robots. These are robots which act in natural environments and fulfill their tasks in an intelligent way. In this thesis, we are concerned with controllers for autonomous mobile robots (or agents) and ways how they can come to intelligent decisions in dynamic real-time domains. Before we set the mission statement for this thesis, we define what *autonomous mobile robots* mean in our understanding and briefly give an overview of what one can understand by the term *intelligent* from an AI perspective.

1.1 Intelligent Robots and Agents: (Mobile) Autonomous Systems

The diversity in the developments of robotic systems nowadays and coupled with the widely spread different software approaches to the control software of these systems, makes it hard to find a common denominator. The tasks for mobile robots are very specific, ranging from assembly robots in production plants to soccer playing robots. Each task has specific demands, each robot therefore has sensors and actuators adapted to the task. The control software must be highly integrated to fulfill the real-time requirements of the environment. This makes it very difficult to establish standards in this field. One approach to create a common software platform is the *Microsoft Robotics Studio* (Microsoft 2006) which comes with driver sofware for many commonly used sensors and actuators and a 3D simulation environment. Another interesting open source project is the *Orocos Project* (which stands for Open Robot Control Software). The European community funded this joint project of the K.U.Leuven in Belgium, the LAAS Toulouse in France, and KTH Stockholm in Sweden. Within this short-term project an open source real-time toolkit, a kinematics, and a Bayesian filtering library were implemented (Orocos 2007). Whether or not these advances will be crowned by success or if the community will accept and use such tools, it nevertheless shows that some efforts are being undertaken to provide standard software tools in this field. Besides researchers and professionals, hobbyists are also building robot systems. Today, many web sites exist which give useful hints on how to build motor controllers, how to derive odometry etc. (e.g. (dmoz.org/Computers/Robotics 2007)). But research still deals with fundamental problems like light-weight energy supply, motor devices, appropriate sensors and actuators on the engineering side. On the control software side some achievements have been made, like robust navigation, and localization, at least with regard to wheeled robots. Research on humanoid robots is still in its infancy though good progress is observable. When it comes to robots acting intelligently in an autonomous way, even more unsolved problems exist.

Bekey (2005) defines autonomy as: "*Autonomy refers to systems capable of operating in real-world environments without any form of external control for extended periods of time*". In this sense he concludes that living systems are autonomous systems, which "[...] *survive in a dynamic environment for extended periods, maintain their internal structures and processes, and exhibit a variety of behaviors (such as feeding, foraging, and mating).*" The difference with robots or agent systems is that these are created by humans. Based on this he defines a robot as "*a machine that senses, thinks, and acts*". The term autonomous robot control seems contradictory as autonomy implies the capability of something to take care of itself, and control is connoted with some form of human intervention. Bekey states that control is needed on several layers in a robotic system and that these control structures exhibit behavior as "foraging" which is connected to autonomy (in the robotic context this could mean that the robot is aware of the fact that it needs to recharge its batteries and returns to the charging station). High-level control "*is required to ensure that the robot does not harm any humans or equipment of other robots*".

According to Murphy (2000), who briefly reviews the history of robots, robots are perceived as anthropomorphic, mechanical, and literal-minded servants. The acceptance of non-anthropomorphic creatures like robot systems existing today is due to the fact that robots are mechanical, and thus wheeled robots are accepted as robots although they are not anthropomorphic. Murphy (2000) defines an intelligent robot as "*a mechanical creature which can function autonomously*", where intelligent means that the robot does not perform tasks mindlessly which is seen as the opposite of factory automation. Autonomy indicates "*that the robot can operate, self-contained, under all reasonable conditions without requiring recourse to a human operator*" and "*that a robot can adapt to changes in its environment* [...] *or itself* [...] *and continue to reach its goal*".

We are aiming at mobile autonomous robotic systems which act in an "intelligent" way. The field of research on mobile intelligent autonomous robots is also called *Cognitive Robotics*. Those systems have been built for over 30 years. Beginning in the 70's with the robot *Shakey* (Nilson 1984) which was the first robotic system to which one can assign the attributes mobile, autonomous, and intelligent, major improvements have been made. One successful application was the Rhino project (Burgard et al. 1998). The robot Rhino was operating over several days in the Deutsche Museum, Bonn, as a museum tour-guide. It operated safely in the very crowded museum, interacting with visitors. These kind of applications are called *service robotics* applications. There are numerous other examples. Pineau et al. (2003) used robots to support elderly people in nursing homes. The robot reminded the elderly to take their medicine and guided them to their examinations. Another intriguing application for mobile autonomous systems is the DARPA

Grand Challenge (Buehler et al. 2007). In the 2006 competition an autonomous vehicle had to drive about 200 miles through the Mojave Desert. DARPA advertised the Grand Challenge with a prize money of 2 million USD. The team from Stanford was able to win the challenge (Thrun 2006; Montemerlo et al. 2006). For other examples of successful robotics applications we refer to textbooks on robotics (e.g. (Arkin 1998; Kortenkamp et al. 1998; Murphy 2000; Bekey 2005; Thrun et al. 2005)).

A bit less spectacular than driving through a desert (and with much less prize money), but nevertheless very interesting and challenging, is the application of soccer playing robots (Kitano et al. 1997; RoboCup 2006) which affords some form of intelligent behavior. The ambitious goal is to be able to win against the FIFA soccer world champion with a team of humanoid robots by 2050. To inspire research on robotics and Artificial Intelligence research there are annual tournaments where research groups compete with their teams against each other. During these competitions promising approaches are evident. The exchange of knowledge of the participating research groups is further speeding up the development of these systems. The soccer application is of special interest as the robots are facing an adversarial dynamic real-time domain. Different leagues are organized in RoboCup which focus on different aspects of research on intelligent robots or agent systems. (We will not go into details here and refer to Chapter 5.2 where we discuss the soccer domain in detail.) Besides the soccer playing activities, other fields of mobile robotics are covered. There are *Rescue Leagues* where the goal is to seek injured people in an urban disaster area, or the service robotics league *RoboCup@Home*, where the robot should fulfill helper tasks in a household.

But the focus for intelligent systems is not restricted to robot systems. With respect to high-level control and decision making of such systems, we also have to keep in mind agent systems in general. Wooldridge and Jennings (1995) give a weak notion of an agent. According to this "[...] *the term agent is used to denote a hardware or (more usually) software-based computer system that enjoys the following properties: autonomy* [...]; *social ability* [...]; *reactivity* [...]; *pro-activeness*". The stronger notion they give addresses, besides the already mentioned attributes, properties that are "[...] *either conceptualised or implemented using concepts that are more usually applied to humans*" (Wooldridge and Jennings 1995) (see also (Wooldridge 2002)). Goodwin gives the following properties of agents in (Goodwin 1995): (1) *successful*, i.e. the agent can accomplish its tasks; (2) an agent is called *capable*, if it possesses all the effectors needed to fulfill the tasks; (3) *perceptive*, which means it can distinguish the salient characteristics of the world it acts in, (4) it is *predictive*, if its model of the world is accurate enough to correctly predict how it can or cannot achieve the tasks; (5) agents are called *interpretive* if they can interpret their sensor readings correctly; (6) *rational* if it chooses to perform commands that it predicts to achieve the task; and (7) *sound*, if it is predictive, interpretive, and rational. Other characteristics assigned to intelligent agents are mentalistic notions, such as *knowledge*, *belief*, *intention* and *obligation* (Shoham 1993), *mobility* (White 1999), *veracity* (Galliers 1988, pp. 159–164), or *benevolence* (Rosenschein and Genesereth 1988, p. 91). The rationality property of an agent or robot can be seen as an *action selection problem*. Maes (1989) formulates the action selection problem as "*how can an agent select 'the most appropriate' or 'the most relevant' next action to take at a particular situation?*"

This draws the picture of what we are concerned with in this thesis. We are dealing with mobile autonomous robot (agent) systems which act in an intelligent way (mobility should not be taken too literally regarding agent systems) in the sense given above and with ways in which intelligent decisions for choosing appropriate actions can be made. There are many different approaches to the action selection and decision making problem. They range from *Behavior-based AI* to *Knowledge-based AI* (we discuss the different approaches and paradigms in the next chapter).

1.2 Methods and Models for Making Them Behave Intelligently

How can we model intelligent behaviors for autonomous robots or agents? The classical approach is from Knowledge-based AI, based on theorem proving (Green 1969). It makes use of a *sense-plan-act* cycle, that is, the robot gathers new sensory information, starts a planning process about its future course of actions, and performs them. With this scheme the robot Shakey (Nilson 1984) planned its tasks. The planner worked on a STRIPS representation (Fikes and Nilson 1971) (we discuss this approach in the next chapter). The decision making works, roughly, as follows. The robot has several actions each with a precondition and an effect. The task is, given an initial state and a goal state, to find a sequence of actions which fulfills the preconditions of the goal state. Again, there is a vast number of publications on planning methods. One such planning approach is *partial order planning* and its derivatives (Penberthy and Weld 1992; Weld 1994). Another approach is to use task nets which are organized hierarchically instead of planning with basic actions. This approach is called *hierarchical task network* (HTN) planning (e.g. (Sacerdoti 1975; Tate 1977; Erol et al. 1994b; Erol et al. 1996; Firby 1996)). Other techniques are so-called *practical reasoning* techniques, where the knowledge is encoded in forms of procedures (recipes) (e.g. (Georgeff and Lansky 1986; Myers 1996)). Relying on a sense-plan-act cycle bares the risk of not being reactive enough (especially in the early times when computation power was comparatively low). Once a plan was constructed it was performed regardless of the current state of the world which might have changed due to long planning times.

To overcome this problem, the plan cycle was simply ignored. This lead to so-called behavior-based systems. One prominent example of a huge number of approaches of Behavior-based AI is the work of Brooks (1991). The behavior is exhibited directly from the sensor readings without an explicit world model representation. The paradigm states that intelligent behavior emerges from this *sense-act* scheme (see also (Arkin 1998) for a thorough treatment of this topic). While this approach works adequately for tasks on lower levels (choosing between, say, localization and collision avoidance) there is doubt that this approach alone is a suitable model for robots interacting in a dynamic environment in an intelligent way as described above. The reactive paradigm clearly has advantages in being able to react quickly to changes in the environment. To combine to pros of both approaches numerous hybrid architectures were proposed thereafter. The idea is to perform the *plan* step asynchronously to a *sense-act* cycle. Examples of hybrid architectures in the literature are (Arkin 1989; Gat 1992; Simmons 1994; Bonasso et al. 1997; Konolige et al. 1997)

Learning techniques can also be deployed sucessfully. To give an example, we can look at the robotic soccer domain. Here, work exists showing that these kind of techniques pay off for the control problems of a mobile robot or agent. Looking at recent proceedings of the *RoboCup*

Symposium (Polani et al. 2004; Nardi et al. 2005; Bredenfeld et al. 2006; Lakemeyer et al. 2007) one notices that a lot of control problems are solved by applying learning techniques. One example for a learning architecture can be found in (Stone 2000). On all layers (motion control, behaviors, decision making) of the described control architecture of a soccer agent in the 2D Simulation league, learning techniques are successfully deployed. Riedmiller and Merke (2002) describe applications of reinforcement learning of autonomous (soccer) agents. While these applications concentrate on the behavior level, some extensions exist to approximate the behavior policy for a team of agents (e.g. (Lauer and Riedmiller 2000)). (Multi-agent reinforcement) learning is an active research field, but to the best of our knowledge, learning methods are currently suited best to learning single behaviors like intercepting a ball or avoiding collisions with other players.

For high-level decision making one problem with learning might be that, in general, these techniques are not adaptive and flexible enough to adopt quickly to a new environmental situation. Thus, like with hybrid robot system architectures, the most promising approach to high-level decision making is probably to also use hybrid techniques here. Two such approaches which take advantage of this idea are the high-level robot programming languages Golog (Levesque et al. 1997) and RPL (McDermott 1991; Beetz 1999). Golog and its descendants, for example, combine the idea of robot programming with planning based on the formal logical language of the situation calculus (McCarthy 1963). RPL follows more a reactive approach (although a projection mechanism for RPL programs exists) and in the latest version, combines programming with learning (Beetz et al. 2004). (We discuss the related work on this field in the next chapter.) Other techniques for decision making concentrate on the uncertainty of sensors, actuators and the environment the robot interacts with. Techniques in this field are decision-theoretic planning (DTP, e.g. (Puterman 1994; Boutilier et al. 1999)) and reinforcement learning (RL, e.g. (Kaelbling et al. 1996; Sutton and Barto 1998)). Based on a utility function and some notion of the uncertainty of its own actions the agent or robot estimates (in the case of DTP) or tries out (in the case of RL) the best course of action in each situation (more details about both techniques are given Chapter 3).

1.3 Goals and Contributions of this Thesis

To summarize, several methodologies, methods and techniques to the control problem of an autonomous robot or agent interacting with its environment in an intelligent way, exist. They range from explicit programming via learning approaches to complete deliberative planning techniques. Of special interest are *adversarial, highly dynamic real-time domains*. A robot is faced with the dynamics and real-time requirements of the world in each real world scenario it is operating in. But the real-time and dynamic requirements we mean here refer to the high-level decision making of the robot or agent. In those domains the robot is moving fast and decisions about which actions to perform have to be taken in fractions of a second. Moreover, there are opponents that try to foil its plans and have contrary goals.[1]

[1]We want to stress that we are not given hard real-time constraints, but real-time constraints. We are not dealing with aircrafts or medical devices where a failure leads to a catastrophe. The time scale for the low-level system here is about 40-50 Hz, high-level control runs at about 10 Hz and slower. See for example (Musliner et al. 1995) for a discussion of real-time AI.

One such domain that we focus on and which inspired several design decisions is the already mentioned soccer domain. It has the aforesaid characteristic. To give an example, a robot dribbling with the ball means it is an adversarial real-time domain. It tries to reach its target position without losing the ball. For the control modules this means that they always have to provide appropriate actuator commands in the needed time frame. If the motion control, for example, cannot keep up with its decision time, the robot might decelerate and lose the ball. As an example for the real-time requirements of the high-level decision making think of a robot staying in front of the ball. The robot deliberates what to do with the ball, evaluating sensor values where the opponent goal keeper might be positioned, if it is possible to score directly, or it makes some complex plan about passing the ball to a better positioned teammate. If all this takes too long, an opponent will simply steal the ball rendering all efforts of the robot about its future actions useless.

This shows that the high-level control of a robot in such domains must be reactive enough to avoid such situations. On the other hand, it seems advantageous to provide some form of deliberation in order to analyze the current situation of the world, think about different alternatives of what to do and choose one which, against the current knowledge background of the robot about the world, is rational: i.e. exhibits intelligent behavior intentionally.

Between the two extremes of the reactive and the pure deliberative paradigm we are aiming at a robot programming language which is capable of integrating the advantages from both worlds. It must be able to react quickly to a new situation and reason about which future courses of actions to take, for example in the soccer scenario, playing a deliberate attack over the left wing. Furthermore, the language for high-level robot controllers in these domains must be capable of providing for some notion of uncertainty. The robot's own perceptions and actions as well as the behavior of the opponents are uncertain and there should be ways to incorporate this. We are thus aiming at a middle ground between reaction, deliberation, including uncertainty of its own actions and being able to account for continuous changes in the world.

Our approach to high-level controllers for robots acting in dynamic real-time domains is founded on the robot programming language Golog (Levesque et al. 1997). During the last decade since it was proposed, many extensions to the original language have been made. With these it is a suitable language for programming the high-level control of robots. Especially in the Cognitive Robotics community this language is accepted and Golog was used in several successful robotics applications for implementing high-level control. One such example is the already mentioned museum tour-guide Rhino (Burgard et al. 1998; Hähnel et al. 1998). The application domains of Golog so far were mostly for service robotics. The typical applications for cognitive robots in the literature are delivery tasks in office environments (e.g. (Simmons et al. 1997)) or similar applications. With our work we want to show that with further modifications to the logic-based approach with Golog it can be applied successfully to the kind of dynamic real-time domains we introduced above, where it is state of the art to use reactive systems.

The crucial point is that logic-based approaches incorporating planning have the drawback that they are computationally more demanding than associating an action to a situation. Regarding deliberation we therefore also aim at a middle ground between full planning and programming. What we propose is to endow the robot with a partially specified control program where the controller based on its background theory fills in the missing details and chooses between open action alternatives. The method we chose is that of decision-theoretic planning in the Golog framework. While the extensions we draw on are known from the literature (Levesque et al. 1997; De Giacomo et al. 1997; De Giacomo and Levesque 1999; De Giacomo et al. 2000; Soutchanski 2001; Boutilier et al. 2000; Grosskreutz and Lakemeyer 2000b; Grosskreutz and Lakemeyer 2001), it is the integration of these extensions into a unified framework that is one of the contributions of this work. We propose the Golog-based language READYLOG as a control language for robots or agents acting in dynamic real-time domains in this thesis. Existing extensions like on-line execution, sensing, using probabilistic projections into the future, or dealing with continuous change have been integrated into one framework. In order to equip the robot with a controller framework allowing for reaction, deliberation (in the sense of projections into the future and decision-theoretic planning) and execution of plans, several further extensions to the known approaches had to be made. In particular, the contributions of this thesis are:

- *Language definition of* READYLOG
  READYLOG integrates existing extensions for continuous change, probabilistic projections, on-line execution, exogenous actions, (passive) sensing, and decision-theoretic planning. We propose a new approach to on-line execution of policies generated with the decision-theoretic forward search value iteration algorithm based on passive sensing. We extend existing passive sensing approaches in the way that the READYLOG controller is equipped with a world model where it can derive data directly without the need to regress fluent values. This, on the other hand, demands an efficient method to progress the internal database. We intregrated a method which is based on the theoretical work of Lin and Reiter (1997).

- *On-line execution and monitoring for policies*
  For executing policies which were calculated by the decision-theoretic planning algorithm it is very important to use execution monitoring. As the planning is done off-line, it might be the case, especially in fast-paced domains, that the world changed in an unpredicted way. The controller must be able to detect discrepancies between planning and real execution. We analyze the approach by Soutchanski (2001) and show several drawbacks of this approach w.r.t. our application. The method we propose with on-line execution of policies circumvents the problems of the former approach. Further, we extend the notion of stochastic actions in our language to restricted stochastic programs that simplifies the specification of the stochastic outcomes of an action and show that our models are correct w.r.t. models known from literature (Reiter 2001).

- *Macro-actions or options in the* READYLOG *framework*
  As an further extension to decision-theoretic planning we define and introduce options in the READYLOG framework. The idea of options is adopted from (Hauskrecht et al. 1998; Precup et al. 1998; Sutton et al. 1999). Options are macro actions in the context of Markov decision processes. With our extension macro actions can be used for decision-theoretic planning in the READYLOG framework. We show an exponential speed up in the planning times when using these macro actions. We propose some more extensions like using pruning to further speed up the calculation of policies.

- *A control architecture for mobile robots*
  As stated above, we want to show that READYLOG can be successfully applied to domains where planning seems to be too demanding. This implies that we have to provide a robot system where READYLOG can be used as the high-level control component. Therefore, some parts of this thesis deals with robotics. The contributions made are a modification to a well-known localization algorithm with proximity sensors. The modification allows a robot to localize in environments with very sparse landmarks. Further, we propose an efficient and robust navigation and collision avoidance algorithm which scales up to ground velocities of over 10 km/h. Finally, as a precise world model is helpful from the viewpoint of high-level decision making, we compare several sensor fusion methods using the problem of ball position estimation in the soccer domain.

- *State space abstractions using qualitative world models and an approach to formalize soccer with* READYLOG
  The expressiveness of READYLOG together with its reasoning capabilities allows for specifying and executing complex behavioral patterns. Usually, soccer robots follow very simple strategies like *intercept–dribble–shoot*. Inspired by human soccer literature, we formalize and adapt soccer strategies derived from human soccer theory in READYLOG (Dylla et al. 2007). One result from this work was that it turnes out that humans generally represent space, distance, and positions on the field in a qualitative fashion.

  Derived from this we developed a qualitative world model for our soccer robots which simplifies the specification of the behavior for the soccer domain as it can be encoded in a more natural (human-like) way. Moreover, the qualitative world model can be used for abstracting the state space underlying the MDP when exerting decision-theoretic planning or options in the soccer domain. As an extension to the options presented in Chapter 4 we present a DT plan library. The idea is to store abstract policies, i.e. those policies where all choices can still be taken, in a plan library. Later, the agent can draw on its plan library and simply pick an existing abtract policy without the need to calculate it again. Such, the agent saves on-line computation time. We show with two examples of the simualted soccer environment how this plan library saves computation time, and, moreover, how macro actions can be encoded this way.

We applied READYLOG to the soccer domain in simulations as well as with a real robot team. At several RoboCup tournaments it showed its applicability and usefulness. Further, READYLOG was applied for controlling game bots in the interactive computer game Unreal Tournament 2004. The low-level control software we present in this thesis proved to be very suitable for soccer applications as well as service robotics applications and showed flexibility and extendability. The control software proved useful as framework for other robotics projects like multi-robot localization (Amiranashvili and Lakemeyer 2005; Amiranashvili 2007) or sound-localization (Calmes et al. 2007). Recently, it was used for a very successful application in the RoboCup service robotics competition (Schiffer et al. 2006a)

1.4 Outline

The rest of this thesis is organized as follows.

Chapter 2 overviews the related work on high-level robot/agent control. We discuss other action formalisms and approaches to the problem of high-level decision making. Here we present the big picture of related work on the fields of high-level robot control and similar areas. Throughout the thesis we will discuss the more specific related work at the end of each chapter.

Chapter 3 introduces the mathematical foundations for this work. We will introduce the model of Markov Decision Processes which we will need throughout this thesis. One can distinguish between model-based (decision-theoretic planning) and model-free approaches (reinforcement learning). Then, we present a brief overview of reasoning with uncertainty, especially Bayes Filter. These become relevant when we describe approaches to localize a mobile robot in its environment. Finally, we give a detailed introduction into the situation calculus and GOLOG with some of its derivatives. This is the action formalism we found our approach to high-level robot programming on.

Chapter 4 presents the language READYLOG, our proposal for a GOLOG-based robot programming language. The focus of this thesis is on dynamic real-time domains. Thus, the language READYLOG integrates aspects from different GOLOG derivatives. It offers features like dealing with continuous change or decision theory. We first present READYLOG's formal semantics before we concentrate on decision theory. The policies (optimized programs) can become invalid quite soon in highly dynamic domains. Therefore, one must be able to detect when a policy becomes invalid. We propose an execution monitoring scheme. Further, we introduce the concept of macro actions in the context of decision-theoretic GOLOG which helps to drastically decrease the complexity of the planning task. Several other extensions are proposed which speed up the calculation of policies. Thereafter, we address the implementation of READYLOG. We show how the READYLOG interpreter works in detail and discuss the implementation of passive sensing, exogenous/sensing actions, and how the high-level control can be connected to the rest of the robot/agent system. We sketch the implementation of the method for progressing the initial database as well. Parts of the contents of this chapter have been previously published in (Ferrein et al. 2003; Ferrein et al. 2004; Ferrein et al. 2005b)

Chapter 5 shows application examples of READYLOG. We start with an READYLOG agent, also called game bot, for the interactive computer game UNREAL TOURNAMENT 2004. We concentrate especially on decision-theoretic optimizations and show the different modeling possibilities within READYLOG with a study of the the "item pickup task". Then we introduce the robotic soccer domain and show why this application domain is of special interest by giving the different characteristics of this domain. We show a READYLOG soccer agent for simulated soccer. With the simulated soccer agents we demonstrate the possibility to use probabilistic projections to come to a decision. Parts have been previously published in (Dylla et al. 2003b; Jacobs et al. 2005a; Jacobs et al. 2005b).

Chapter 6 shows soccer applications of READYLOG with real robots. Besides the READYLOG application, the focus of this chapter is on the robot system as a whole. In particular we show that a well integrated software system is needed before one can think about high-level control with READYLOG. First, we overview the hard- and software of our robot soccer team "AllemaniACs", before we describe our approach to navigation and localization in detail. We also present an account to global sensor fusion to enhance the world model of the robot here. At the end of this chapter, we briefly prospect the usefulness of READYLOG in the cognitive robotics area with a small service robotics example. Previously, material from this chapter were published in (Dylla et al. 2003b; Ferrein et al. 2004; Ferrein et al. 2005b; Jacobs et al. 2005a; Jacobs et al. 2005b; Ferrein et al. 2006; Strack et al. 2006).

Chapter 7 presents an alternative approach to the problem of state space abstractions. For soccer we investigated how humans represent strategies and how this could be helpful for the behavior specification of a robot. It turns out that READYLOG is very well suited to formalize soccer strategies and that humans make extensively use of qualitative notions, for example distances (near or far) or regions on the field (on the left front half of the field). From these observations we derive a qualitative world model for the soccer domain which was applied to our soccer robots. The behavior specification is simplified and much more "natural". For READYLOG this qualitative world model can be used to abstract from an infinite state space and to make techniques presented in Chapter 4 applicable. We introduce a method where policies are left in an abstract way rather than calculating the optimal values at once. These stored policies form a library of already instantiated plans which the agent can draw on with a dramatic saving in on-line computations. We present two examples from RoboCup's simulation league. Parts of the material shown in this chapter were published before in (Ferrein 2004b; Dylla et al. 2005; Schiffer et al. 2006b; Dylla et al. 2007; Böhnstedt et al. 2007).

Chapter 8 concludes this thesis with a summary and an outlook on future work.

More material, especially on the robot soccer team "AllemaniACs" can be found on the web (Schiffer et al. 2006), in (Ferrein 2004a; Fritz 2004; Ferrein and Lakemeyer 2005; Ferrein and Lakemeyer 2006; Calmes et al. 2006), and in several official Team Description Papers for participating at RoboCup tournaments (Dylla et al. 2003a; Ferrein et al. 2005a; Ferrein et al. 2006; Ferrein et al. 2007a; Ferrein et al. 2007b). Some of the material covered here was also published in several Diploma and Master's Theses before: (Jansen 2002; Fritz 2003; Hermanns 2004; Strack 2004; Schiffer 2005; Jacobs 2005; Böhnstedt 2007). I would like to thank explicitely for the support and the contribution.

Chapter 2

Related Work

In this chapter we review approaches related to our work. Particularly, we discuss work in the field of reasoning about actions, (Section 2.1), the robot programming language Golog and its derivatives (Section 2.2) and control architectures for autonomous robots or agents in general (Section 2.3). We conclude with justifying our approach against the context of related approaches to reasoning about actions.

2.1 Reasoning about Action and Change

For deliberation several different approaches to the problem exist, like *practical reasoning*, *hierarchical task nets*, and *reasoning about action approaches*. In this section, we will review hierarchical task networks and the model of Markov decision making, and several logical calculi for reasoning about actions and change. As an example for a practical reasoning approach we discuss the language 3APL.

2.1.1 Hierarchical Task Networks

In contrast to classical planning methods where a goal is to be achieved with a sequence of primitive action, the idea of hierarchical task network (HTN) planning (Sacerdoti 1974) is to accomplish a task, given a task network. This includes decomposing compound tasks and resolving conflicts which might occur between different tasks. HTN planning can be seen as the practical approach to letting a robot do something in reality, while in the early days, classical planning methods (like STRIPS planning, cf. Section 2.1) were computationally prohibitive for practical applications. Another motivation for HTN planning was to close the gap between AI planning techniques, project management and scheduling techniques known from operations research. Early HTN planning systems were NOAH (Sacerdoti 1975) or NONLIN (Tate 1977). While HTN planning was used in practice for a long time, it was only until the mid 1990's that a semantics for HTN was defined (Erol et al. 1994a; Erol et al. 1994b). More recent HTN planning systems are SIPE-2 (Wilkins 1990) or SHOP and SHOP-2 (Nau et al. 2003).

Recent trends seem to go in the direction to combine HTN formalisms with machine learning. In (Ilghami et al. 2002; Ilghami et al. 2005) the authors propose a learning system for HTNs, where a domain expert solves task nets giving examples to the learner. The learner now generalizes based on the training examples from the human expert and can solve similar tasks in a better way.

Nejati et al. (2006) propose a similar approach. The difference is that they do not assume that the HTN is given in advance. Another example is (Belker et al. 2003). Here, a robot learns navigation tasks based on a HTN representation of sub-tasks. The sub-tasks or actions were of the kind *turnTo* or *moveForward*, and compound tasks were *ApproachPoint* and *Goto*. They tested this approach on a Pioneer II and the B21 Rhino. They report on an increasing performance compared to the original navigation software of Rhino with the Dynamic Window Approach (Fox et al. 1997). One application example from robotic soccer is (Obst and Boedecker 2006). Team coordination and strategy selection tasks are solved applying HTN techniques. The ideas of HTN were also incorporated into other reasoning frameworks. Baral and Son (2000) define task nets and task decomposition in the ConGolog framework which we describe in Section 2.2. A similar approach is followed by Gabaldon (2006). He defined tactical and strategical tasks for military applications in the form of HTNs in the ConGolog framework.

2.1.2 Incorporating Uncertainty: Markov Decision Processes

Markov decision processes (MDPs) (Bellman 1957; Bertsekas 1987; Puterman 1994) are a natural formalism to describe stochastic dynamic systems. It is the underlying formalism to decision-theoretic planning (DTP) and reinforcement learning (RL) (e.g. (Boutilier et al. 1999; Sutton and Barto 1998; Kaelbling et al. 1996)). The world is described by discrete states. Performing actions will lead an agent from one state to another with a certain probability. To each state a real-valued reward is assigned. The planning problem is to find an optimal action for each state which maximizes the long-term reward. Thus, a policy is generated which for each state provides the optimal action w.r.t. to the optimization criterion. Solution techniques like value iteration and policy iteration are well-studied for cases where the transition model between states is known. If the transition model is not known in advance, learning techniques like Q-Learning can be applied (Watkins 1989) to find an optimal policy by observing the outcomes of the deployed actions.

Standard techniques for solving MDPs require an explicit state representation. This has the drawback that for solving real-world problems the standard techniques, which rely on dynamic programming techniques (e.g. (Bellman 1957)), have to iterate over all states; the problem may become computationally intractable (though there are modifications like asynchronous dynamic programming algorithm or modified policy iteration that do not need to cover the whole state space (cf. e.g. (Bertsekas 1987; Bertsekas and Tsitsiklis 1996; Puterman and Shin 1978)). To overcome this problem several methods have been proposed to relax the original problem by abstracting the state and/or action space, decomposing the problem into sub-tasks (Parr and Russell 1997; Precup et al. 1998; Hauskrecht et al. 1998; Sutton et al. 1999), or use factored logical representations for the state space (Boutilier et al. 1995; Dean and Givan 1997; Boutilier et al. 1999; Boutilier et al. 2000; Boutilier et al. 2001; Feng and Hansen 2002).

Among the approaches to decompose MDPs into smaller sub-problems one has to examine the work by Precup et al. (1998). They propose options, sub-MDPs, with an initiation and a termination set of states. As long as the agent is in one of the initiation states it can perform the policy calculated for the option. The agent now follows the policy which was calculated for the option until the option terminates in one of the termination states. Applying options constitutes a semi-Markov decision process (Sutton et al. 1999) as the duration of a primitive action and an option is different. With a modified state transition and reward model, convergence is still guaranteed. Similar to the option approach, (Hauskrecht et al. 1998) define hierarchical macro-actions for MDPs.

For each macro the state space is decomposed into an entrance and exit periphery which is similar to the initiation and termination states of options, though these peripheries are not defined over the whole state space. A macro-action is formally defined as a sub-MDP. For solving the original problem, both basic actions as well as macro-actions can be used together. Another hierarchical approach to the decomposition of MDPs is the approach of Parr and Russell (1997). They propose a nondeterministic finite-state machine, called hierarchical abstract machines (HAMs), which controls what action shall be executed in which partition of the state space. The machine can take one of four different type of states: *action*, which executes an action, *call* which calls another machine as a subroutine, *choice* which nondeterministically selects the next machine state, and *stop* which halts execution of the current state and returns to the previous one. HAMs can also be applied to Q-learning with much faster learning rates. The MAXQ-Q-learning method (Dietterich 2000) assumes that the programmer can identify useful sub-goals and sub-tasks for the application. Based on these tasks and goals the original MDP is divided into sub-MDPs. Each sub-task ends when its sub-goal is reached. Based on the task decomposition the value function of the original MDP is divided as well. The solution of the original MDP is a hierarchical policy which consists of a set of policies or actions, one for each sub-task. By this decomposition MAXQ learns a representation of the value function and it turns out that this method learns faster than Q-learning.

Another approach is to factor the state space with a propositional representation and with it the reward function. A survey of factored MDPs is given in (Boutilier et al. 1999). One such representation is based on dynamic Bayes net (Dean and Kanazawa 1990). For two stages of the MDP a Bayes net is constructed, the transition probabilities are stored in conditional probability tables. Other representations proposed are for example, decision trees (Quinlan 1993), or algebraic decision diagrams (ADDs, (Bahar et al. 1993)). Another representation is probabilistic STRIPS (Kushmerick et al. 1995), which we describe below. A state-of-the-art MDP solver is SPUDD (Hoey et al. 1999), which can solve MDPs with hundreds of millions of states representing the value function with a logical description which involve only hundreds of distinct values. Interestingly, Soutchanski (2003) compared DTGOLOG (described below) with SPUDD and shows that DTGOLOG can solve given examples problem instances faster than SPUDD. Soutchanski (2003) notes that the policies on the reachable state space are identical though the values can be compared only in a qualitative fashion.

Another representation is that of first-order MDPs (FOMDPs) instead of the propositional approaches discussed above. One such method for factoring the state space with first order logic is proposed with DTGolog (Boutilier et al. 2000). An MDP is represented implicitly by a situation calculus basic action theory and the MDP is solved by a forward search value iteration algorithm which avoids state enumeration (see below for a detailed discussion of the situation calculus and DTGolog). Another technique for solving FOMDPs was proposed by (Boutilier et al. 2001) as an extension of the propositional decision-theoretic regression (DTR, (Boutilier et al. 2000)). The advantage of the latter approach is that neither action nor state space have to be enumerated explicitly. DTR represents the value function of the MDP as a first-order formula and based on the stochastic basic action theory of the situation calculus an optimal solution can be calculated with the *symbolic dynamic programming* algorithm proposed in (Boutilier et al. 2001). The problem with this approach w.r.t. the applicability in real-world domains, is to efficiently simplify the formulas at each stage of the dynamic programming. Between approaches like DTGolog or symbolic dynamic programming which both are based on the situation calculus one has to mention Poole's independent choice logic (ICL, (Poole 1997)). Based on facts and a choice space representing the action alternatives he defines a selector function using a possible world semantics. The consequences of the choices made by nature are represented as acyclic logic programs. In his paper he shows how decision-theoretic problems can be formulated in ICL, either in the single or multi-agent case. He further shows how other formalisms can be embedded.

For the sake of completeness, we briefly mention POMDPs. The generalization of MDPs are partially observable MDPs (POMDPs). Here, the assumption that the agent can observe with certainty the state it is in after performing an action, is dropped. Instead the agent has a belief about the state it is in, represented by a probability distribution over possible successor states. A survey on POMDPs is given in (Monahan 1982). As POMDPs are not relevant for this work, we only give some references to related literature (e.g. (Sondik 1971; Sondik 1978; Lovejoy 1991; Cassandra et al. 1994; Barto et al. 1995; Parr and Russell 1995; Boutilier and Poole 1996; Geffner and Bonet 1998)).

2.1.3 The Situation Calculus

The situation calculus was proposed by McCarthy in 1963 and is the earliest approach to formally reasoning about actions. It is a logical first-order language for representing dynamically changing worlds. The world is represented by situations which can be changed by performing various actions. Fluents, which are relations or functions with a situation term as last argument, describe what is true or false in the world w.r.t. a situation. There a distinguished binary function $do : situation \times action \rightarrow situation$ exists which denotes the situation that results by performing actions. A special constant S_0 describes the world in the initial situation. For each action there exists an axiom $Poss(a(\vec{x}), s) \equiv \Pi(\vec{x}, s)$ which denotes the precondition of an action, i.e. when the action a is applicable, and an effect axiom, which describes how the action changes the world in terms of fluents.

One of the problems with the causal laws (effects) in a domain description is that they do not describe the non-effects of an action. For example, picking up an object does not change its color. This problem is referred to as the *frame problem*. The number of frame axioms describing the non-effects of an action is usually much larger than the axioms describing the effects. A general solution to this problem was found by Reiter (1991). He proposes to derive from the effects of an action so-called *successor state axioms* for each fluent which — with some further assumptions like a completeness assumption and a unique names assumption for action — describe both, the effects and the non-effects of the actions. In the next chapter we present his approach in detail. Another problem in the context of reasoning about actions is known as the *qualification problem*. It refers to the impossibility of listing all action preconditions required for a real-world action to have its intended effect. Finally, in the context of reasoning about actions the *ramification problem* is known. It addresses the problem of taking also the indirect consequences of an action into account. These problems are discussed in (McCarthy and Hayes 1969; Ginsberg and Smith 1988; Lin and Reiter 1994; Lin and Reiter 1994; Thielscher 2001).

In (Pinto and Reiter 1993; Pinto and Reiter 1995) Pinto and Reiter introduce a notion of time to the situation calculus. A special fluent $start(s)$ exists which denotes the starting time of situation s. With the formalization of a time line it renders easy to introduce durative actions. Each action a is split into a $start\text{-}a$ and $end\text{-}a$ action. The duration of the action can then be determined as the time difference $start(do([\ldots, end\text{-}a])) - start(do[\ldots, start\text{-}a, \ldots, end\text{-}a])$.

Another key for an expressive action logic is that it has to be able to deal with incomplete initial knowledge. To gather information, sensing must be incorporated. As sensing changes the knowledge of a robot about the world, this issue is related to representing knowledge in the situation calculus in general. To represent the knowledge of an agent Moore (1985) introduced a special fluent $K(s, s')$ into the situation calculus. A solution to the frame problem for knowledge is due to (Scherl and Levesque 1993; Scherl and Levesque 2003). K can be seen as an accessibility relation between situations. So, an agent knows φ in situation s if φ holds in all s' accessible via $K(s', s)$, i.e. $Knows(\varphi(s)) \overset{def}{=} \forall s'.K(s', s) \supset \varphi(s')$. Other useful definitions w.r.t knowledge are $KWhether(P, s)$ denoting whether a fluent P is known or not, and $Kref(\tau, s)$ denoting if an instance of τ is known. Special *sense fluent axioms* connect fluents to be sensed to actions. To express minimal knowledge of the agent, Lakemeyer (1996) and Lakemeyer and Levesque (1998) introduced the concept of *only knowing* into the situation calculus. It means that the agent knows a a set of sentences Φ and nothing more. To express only knowing, they enrich the situation calculus with a possible-world semantics.

Especially when dealing with sensors which, in general, are imperfect, one needs to formulate the belief of the agent or robot. Facts about the world gathered by sensors are not facts which are unconditionally true or false, they can be only seen as beliefs. They are true or false to a certain degree, depending on the quality of the sensors. Beliefs are thus associated with a likelihood (or a probability). In (Bacchus et al. 1995; Bacchus et al. 1999) a formal account to deal with the belief of an agent in the situation calculus is given. In a more recent work Gabaldon and Lakemeyer (2007) develop the language $\mathcal{ESP}$, an extension of $\mathcal{ES}$. $\mathcal{ESP}$ is enriched with a notion of belief and an account to noisy sensors. The advantage is that no second order definitions are needed. We introduce the language $\mathcal{ES}$ in the next section. Similar to the fluent K they introduce a fluent p which denotes the likelihood of a formula φ being true in a situation s. The belief of the agent $Bel(\varphi, s)$ that φ is true in situation s is the weight of the possible worlds where the agent thinks that φ holds normalized by the weight of all possible worlds. Other approaches like (Pinto et al. 2000) take another direction to integrate probability theory into the situation calculus. Here, the goal is not to formalize the belief of the agent but to express non-deterministic actions. The approach is to assign non-determinstic effects to an action and associate it with a probability. With a special transition function the different alternatives at non-deterministic choice points can be evaluated and their successor situation can be determined.

A large body of work exists on the situation calculus. We therefore further refer to (Levesque et al. 1998; Pirri and Reiter 1999) and the book of Reiter (Reiter 2001) for more information about the situation calculus.

2.1.4 $\mathcal{ES}$: The Situation Calculus without Situations

In (Lakemeyer and Levesque 2004; Lakemeyer and Levesque 2005) Lakemeyer and Levesque present the logic $\mathcal{ES}$ which can be seen as a situation calculus without situation terms. The justification for introducing another formalism is that proving mathematical properties of the calculus becomes easier. The main reason for this is that the semantics of $\mathcal{ES}$ is not defined axiomatically as is in the situation calculus but semantically. As the situation calculus, $\mathcal{ES}$ distinguishes between fluents and actions, but has no terms of sort situation. Additionally, it has standard names (e.g. (Levesque and Lakemeyer 2001)) and it is distinguished between fluent and rigid predicate symbols (rigid predicates correspond to situation-independent predicates in the terminology of the situation calculus). As situation terms are dropped they introduce a $\Box$ operator where $[t]\alpha$

means "α holds after action t" and $\Box\alpha$ means "α holds after any sequence of actions". Further, $Knows(\alpha)$ means "α is known" and $OKnows(\alpha)$ means "α is all that is known". The semantics of $\mathcal{ES}$ is based on a possible-world semantics. There exists a world w which determines which fluents are true and an epistemic state e which determines what the agent knows initially. Further, there exists a σ which represents an action sequence. Formulas are evaluated relative to a model $M = \langle e, w \rangle$. The semantics is defined w.r.t. model M and σ. $\mathcal{ES}$ has an direct account to incorporating sensing results, knowledge and only knowledge (the logic $\mathcal{OL}$ (Levesque 1990; Levesque and Lakemeyer 2001) is contained in $\mathcal{ES}$ and thus Lakemeyer and Levesque are able to apply previous results like the *Representation Theorem* (Levesque and Lakemeyer 2001) also to this logic). In (Claßen and Lakemeyer 2006) extensions (like ASK and TELL) of knowledge-based Golog programs are proved. Lakemeyer and Levesque (2004) define the basic action theory with the solution to the frame problem as proposed in (Reiter 1991) in the logic $\mathcal{ES}$ and reprove the Regression theorem in a very compact way. Lakemeyer and Levesque (2005) show with a second-order extension for $\mathcal{ES}$ that the semantics of this logic is expressive enough to handle basic action theories and the Do operator known from GOLOG (see below). This allows for defining the language GOLOG in $\mathcal{ES}$.

Recently, Ziegelmeyer (2006) defined a transition semantics and the semantics for the decision-theoretic extensions of GOLOG in $\mathcal{ES}$ and implemented a prototypical interpreter. Other directions follow lines to integrate state of the art planning techniques into GOLOG based on the $\mathcal{ES}$ semantics (Hu 2006; Claßen et al. 2007). The idea is to embed ADL subsets (Pednault 1989) into GOLOG. These subsets are those defining the semantics of PDDL (Fox and Long 2003), a language standard to define planning problems.

2.1.5 Other Approaches to Reasoning about Action and Change

STRIPS

The Stanford Research Institute Problem Solver (STRIPS) (Fikes and Nilson 1971) is nowadays called the classical approach to problem solving. It was applied to the robot Shakey (Nilson 1984). The idea of STRIPS is the following: "*The task of the problem solver is to find some composition of operators that transforms a given initial world model into one that satisfies some stated goal condition*" (Fikes and Nilson 1971). The operators denote the actions of the robot and are modeled with preconditions which have to be fulfilled in order to be able to apply the action, and has several effects on the world. The preconditions and effects are formalized in propositional logic. So-called add- and delete lists keep track about which facts have to be added or deleted to or from the world model, resp. A forward-chaining algorithm was used to fulfill the preconditions in the goal state by applying operators such that no unsatisfied literals remain. The result is an action sequence leading from the initial state to the goal state.

Representing a plan as an action sequence is in a sense a strong commitment. Often several possibilities to reach a goal exist. Plans can be represented with partially ordered operators (Sacerdoti 1975; McAllester and Rosenblitt 1991). Each linearization of a partially ordered plan is a solution to the planning problem. STRIPS-based partial order planners are for example UCPOP (Penberthy and Weld 1992; Weld 1994), or BURIDAN (Kushmerick et al. 1995). These planners use extensions of STRIPS. With ADL Pednault (1989) introduces conditional effects to STRIPS, PSTRIPS (Kushmerick et al. 1995) allows for modeling stochastic effects. These systems use a backward search strategy as opposed to the original approach. The so-called regression planners begin their search in the goal state and search backward to fulfill the effects of the initial state (Waldinger 1977). Lin and Reiter (1995) formally account STRIPS and also discuss its relation to the situation calculus. Some interesting working notes of John McCarthy about the relationship between STRIPS and the situation calculus can be found in (McCarthy 1985).

The Event Calculus

The Event calculus was introduced by (Kowalski and Sergot 1986) for reasoning about events and their effects and was originally used for database applications (Kowalski 1992). The basic idea is to state that fluents are true at a particular point in time, if an action occurred at an earlier time-point initiated it and did not terminate it in the meantime. The event calculus is said to be narrative-based, which means that a time structure is assumed in which statements about when actions occurred are incorporated (Shanahan 1997). Several axiomatizations in terms of classical, modal or specialized logics exist. We refer here to the classical logic axiomatization given in (Miller and Shanahan 1999).

According to Miller and Shanahan (1999) the event calculus is a sorted predicate calculus with equality with sort action, sort fluents, sort time point, and domain-dependent objects. There are the predicates $Happens$, $HoldsAt$, $Initiates$, $Terminates$, and $<$. $Happens$ describes at which time point an action occurs, $HoldsAt$ denotes that a fluent is true at a certain time point, $Initiates(A, F, T)$ and $Terminates(A, F, T)$ are predicates which express that an action A at time T initiates/terminates the fluent F. "$<$" is a standard order relation for time.

The formalism of the event calculus is able to express nondeterministic effects of actions. Further, events can occur concurrently. It is distinguishes between cumulative and canceling effects. Two or more effects are called cumulative if the simultaneous occurrence of events imposes an effect which none of them has alone, and canceling if the effect of the occurrence of one event prevents the second event to have the effect which it would have without the other event. This results from the axiomatization as different actions may refer to the same time point to change a fluent value. With the underlying time line durative actions can be modeled easily. Hierarchical planning is achieved by compound actions (conditionals, loops, procedures). In (Shanahan 1990) Shanahan extends the calculus to handle continuous change.

Planning in the event calculus is an abductive reasoning task. Circumscription is used to explain the consequences (what holds at a certain time point) by means of the predicate $Happens$ (which action occurred) (see e.g. (Lifschitz 1994) for a detailed discussion on circumscription). Circumscription as goal completion for Horn formulas yields a solution to the frame problem (McCarthy 1980). By adding actions to the background theory, which are known to have happened at a certain time point, a seamless integration of robot programming and planning can be achieved. Planning can also be seen as partial order planning and in (Shanahan 2000) an algorithm similar to UCPOP (Penberthy and Weld 1992) is given.

Shanahan and Witkowski (2001) give an example of how the event calculus can be used for controlling a Khepera robot and how a robot is programmed with the calculus for a navigation task. It is based on the abductive event calculus planner introduced in (Shanahan 2000). Planning and sensor data assimilation can be regarded as abductive reasoning tasks. If the results of sensor data assimilation conflict with the the current plan, re-planning is initiated. The use of compound actions allow for hierarchical planning.

A recent textbook (Mueller 2006) treats the event calculus in-depth. There are some papers which relate the event calculus to the situation calculus (see e.g. (Belleghem et al. 1997)).

Fluent Calculus

The Fluent calculus is an approach to reasoning about actions and change similar to the situation calculus. The fluent calculus is a sorted logical language with sorts actions, situations, fluents, and states. Derived from the situation calculus, it deals with one of the obvious drawbacks of this calculus.

The observations made in (Thielscher 1998) are that determining a fluent value by regressing the action history up to the initial situation is not the most efficient way to derive a fluent value. Instead, Thielscher proposes to represent what holds in the world by a world state instead of implicitly by an action history. The application of a single state update axiom (Thielscher 1999) is sufficient to infer how an action changed the world state. With this, he gives an elegant solution to the inferential Frame Problem, i.e. the problem of inferring the non-effects of the action currently applied. Central to his approach is the *state update axiom* for actions. In a nutshell, Thielscher switches the roles of actions and fluents. While in the situation calculus a successor state axiom is devised for each fluent, a state update axiom is devised for each action in the fluent calculus. This requires that fluents are reified as terms in the logical language. With this approach to state update axioms a world state can be described as the conjunction of positive and negative fluent formulas w.r.t. a given situation term. To be able to handle incompletely described initial situations it is required to reify also the conjunction of fluent terms. He therefore introduces the connective "$\circ$" with the properties of associativity, commutativity and unit element. His representation requires the extension of the unique names assumption for fluents also to states. Thielscher (1999) points out that introducing an equational theory AC1 and the extended unique names assumption at first sight raises the complexity of the theorem proving task. But under the common assumption that there are more fluents than actions in the domain description these extra costs pay off.

With FLUX (Thielscher 2002a; Thielscher 2002b; Thielscher 2005), which stands for FLUent eXecutor, Thielscher introduces a kind of run-time system for the fluent calculus. Constrained logic programs encode agents' tasks based on the so-called FLUX kernel which implements the state update axioms. The papers (Thielscher 2002a; Schiffel and Thielscher 2005; Schiffel and Thielscher 2006) deal with the connection between GOLOG (see below) and FLUX and show how the semantics of GOLOG programs can be transferred to the Fluent Calculus. Several other extensions known from other calculi have been transferred to the Fluent calculus (e.g. (Thielscher 2000; Großmann et al. 2002; Fichtner et al. 2003; Martin 2003)).

At the 2006 AAAI General Games Playing Project Competition (Genesereth et al. 2005; GGP 2006) Schiffel and Thielscher demonstrated the strength of FLUX by winning the competition (Schiffel and Thielscher 2007). The idea of General Game Playing is that from a description of a game a computer program is derived which develops a winning strategy without human interaction.

3APL

Hindriks et al. (1999) presented the agent programming language 3APL. They base their approach explicitly on the *intelligent agent metaphor*: intelligent agents have a complex mental state, they act pro-actively and reactively, and have reflexive or meta-level reasoning capabilities. Thus, agents in their approach are equipped with a belief base which is represented in an arbitrary logical language. In the line of BDI agent architectures (Bratman 1987) their model is based on an agent having a set of beliefs, intentions, or desires to fulfill its goals. Formally, they do not distinguish between them. The basic ingredients of 3APL are basic actions, achievement goals, and test goals which are classified as basic goals in the 3APL terminology. Complex goals are composed from basic goals with sequential composition or are connected via nondeterministic choices including loops and conditionals. They found their approach on logic programming, and the reasoning paradigm is that of practical or means-end reasoning (e.g. (Georgeff and Lansky 1986)).

Thus, they introduce so-called *semi-goals* from which the practical reasoning rules are derived. These are constructed from basic actions, achievement goals, tests, complex goals, and first-order formulas. Practical reasoning rules are of the form $\pi \leftarrow \varphi \mid \pi'$ and mean that if the agent adopted some goal or plan π and believes φ it may consider to adopt π' as a new goal. This can be seen as an extension to recursive procedures known from imperative programming languages. Hindriks et al. distinguish between four types of rules, *failure*, *reactive*, *plan*, and *optimization rules* which are applied in this priority. They formalize *intelligent agents* as agents consisting of a goal base, a belief base, and a rule base.

These rules look like ordinary logic programs. The authors claim that an agent endowed with these kind of rules must be seen differently. As the agent can modify its goals by means of the rules given, it is self-modifying its code and this makes the difference to ordinary logic programs. 3APL has a formal operational transition semantics. Derivation rules define the semantics of the execution of (composite) plans, basic actions, tests, and the application of practical reasoning rules. As the agent, when executing actions, has to decide on a goal to achieve and which rules to apply, the authors separate control structures from the object level on which plans are defined. They therefore introduce a meta language which defines execution rules for action selection where they distinguish between selecting goals and selecting rules. The action *selex* selects a goal from the goal base which is executable in the current mental state of the agent, while *selap* chooses a rule from the rule base. Further, they define the execution of as many goals as possible with the action *ex*, and action *apply* means to apply as many rules to as many goals as possible. Finally, they define derivations for *assignment for program and rule variables*, *tests for goal and rule terms*, *sequential composition*, *nondeterministic action choice*, and *nondeterministic repetition*. An embedding of ConGolog (see below) in 3APL (Hindriks et al. 2000) shows the close relationship between both languages.

(Some) Further Approaches to Reasoning about Actions

Gelfond and Lifschitz (1993) propose the language $\mathcal{A}$. It is a propositional language for reasoning about actions and can be seen as a propositional fragment of Pednault's ADL (Pednault 1989). The ontology of the language is based on fluents and actions. It distinguishes between two kinds of propositions namely value and effect propositions. The former specifies a fluent value at a specific situation, the latter describes the effect of performing an action. The semantics of executing actions is described by extended logic programs. Son and Baral (2001) introduce the notion of knowledge into the language. Their extension distinguishes between ordinary and knowledge states where an ordinary state consists of a set of fluents, and a knowledge state consists of a state and a set of states. They introduce so-called k-propositions that state which fluents become true after executing a sensing action. A transition function defines the semantics of their dialect by mapping from actions and knowledge states to knowledge states. They prove that their formalization is sound w.r.t. the axiomatization of knowledge given in (Scherl and Levesque 1993). A probabilistic extension is defined in (Baral et al. 2002). The successors of $\mathcal{A}$ are the languages $\mathcal{B}$ and $\mathcal{C}$ (Gelfond and Lifschitz 1998) (besides the languages $\mathcal{P}$-$\mathcal{R}$ presented in this paper, which deal with temporal projections). New propositions about static and dynamic laws are introduced. In the latest variant $\mathcal{C}+$ Giunchiglia et al. (2004) distinguish between *static laws* in the form of $\mathbf{caused}\ F\ \mathbf{if}\ G$, which reads if G holds than F is true, *dynamic action laws* when F in the example above is an action formula, and *dynamic fluent laws* in the form $\mathbf{caused}\ F\ \mathbf{if}\ \top\ \mathbf{after}\ c$. The latter means that F is true after action c was executed. They further define abbreviations for exogenous events and inertial laws to account for the frame problem. Sandewall proposes the *Cognitive Robotics Logic* (CRL) in (Sandewall 1998) based on earlier work in (Sandewall 1995). He presents a meta-theory for reasoning about actions. The language allows for expressing durative actions, composite actions, nondeterministic actions, nondeterministic timing of actions and their effects, continuous time and piecewise continuous fluents, imprecise sensors and actuators, and action failures. Similar to CRL, the temporal action logic TAL (Doherty et al. 1998; Kvarnström et al. 2000) makes use of a surface language representing narratives, and a base language allowing the agent to reason about narratives. Logical entailment is based on circumscription. The language TAL is also applied to deliberative tasks for unmanned aerial vehicles (see below).

2.2 The Robot Programming Language Golog

Golog. Golog stands for al*GOL* in l*OG*ic. Levesque et al. (1997) proposed this robot or agent programming language. The idea of Golog is to combine agent programming with reasoning in an efficient way. Golog has its formal foundations in the situation calculus. It offers constructs known from imperative programming languages like conditionals, loops, or recursive procedures. The strength of Golog is, moreover, that it is capable to reason about action and change. The semantics of Golog is defined via a macro $Do(\sigma, s, s')$. The program σ is translated by macro expansion into a situation calculus formula. Program synthesis is done by proving that the situation calculus basic action theory entails that the program reaches the specified goal situation. As a side-effect this constructive proof yields an executable program where nondeterministic choices for actions or arguments of actions are instantiated. It turned out that it can be applied successfully as a high-level language for encoding robot controllers, as we show below. Nevertheless, the original Golog lacks expressiveness for many problems arising in practice. During the last decade many useful extensions have been proposed, most of which we will discuss next.

ConGolog. De Giacomo et al. (2000) propose an alternative formal semantics for Golog, a transition semantics. They define a predicate $Trans(\sigma, s, \delta, s')$ which transforms the ConColog program σ in situation s to a successor configuration $\langle \delta, s' \rangle$, the remaining program after the execution of the first action in σ with the resulting situation s'. A predicate $Final$ exists which denotes if a program legally terminates. The transition semantics allows to define the concurrent execution of programs. De Giacomo et al. introduce so-called high-level interrupts. Triggered by a condition, the actual program execution is suspended and the program associated with the condition is executed until the condition becomes false again. Then the suspended program is continued. They further give a formal account of so-called exogenous actions, actions which are beyond the control of the agent. These actions or events are imposed by the environment, and the agent can properly react to these events. The transition semantics demands a reification of programs, i.e. programs have to be defined as functions in the logical language of the situation calculus.

sGolog. Lakemeyer (1999) proposes sGolog as an approach to deal with sensing results in the Golog framework. sGolog, like Golog and ConGolog, is off-line, i.e. program synthesis is not interleaved with program execution. Therefore, one needs a special treatment of possible results of sensing actions. Similar to the idea of a policy known from MDP theory, Lakemeyer provides a program branch for each possible sensing result, which reacts appropriately to the sensed value. He introduces a statement $branch\_on(\varphi)$ which allows the user to define so-called condition action trees (CAT) on the condition φ. CATs, formally defined in (Lakemeyer 1999), provide alternative programs w.r.t. the sensing result of φ. For his formalization he draws on (Scherl and Levesque 1993; Levesque 1996) and defines CATs and $branch\_on$ in terms of the fluent $KWhether$ and a special function $SF(a, s)$ which represents the fluent value when the sensing action a is executed in situation s. Note that sGolog is an off-line Golog dialect. It was the first approach to formally introduce sensing in Golog. Later approaches to sensing are based on on-line interpretation of Golog programs (Sardiña 2001).

Concurrent Temporal Golog. Finzi and Pirri (2004) propose a concurrent temporal Golog dialect suitable for constraint-based interval planning (see e.g. (Smith et al. 2000)). They integrate temporal aspects from the situation calculus and interleaved concurrency similar to ConGolog (but use an evaluation semantics). Several processes exist which need to be executed concurrently. Each process has an execution time with a start and an end time on a continuous time line. Constraints are legal relations between processes, such as A *before* B, A *after* B, or A *meets* B (A ends in the moment when B starts). Golog now fills in the missing details of a partially specified candidate plan, considering the process constraints. The motivation for constraint-based interval planning is that a robot has to perform multiple tasks, many actions like controlling the camera and actuating the motors have to be performed in parallel. Interval planning helps to order the different activities of the robot to achieve the goal.

IndiGolog. IndiGolog (De Giacomo and Levesque 1999) overcomes one of the central drawbacks of the original Golog approach w.r.t. the applicability for realistic scenarios. It makes use of the transition semantics introduced by (De Giacomo et al. 2000) in the ConGolog derivative. While ConGolog was still off-line, IndiGolog aims at on-line execution of robot programs. The interpreter directly commits to action choices made, by executing them. This is a fundamental difference to Golog, where actions are executed only after the whole program/plan has been synthesized. The incremental execution enables the agent to gather new world information. In (De Giacomo et al. 2001) sensing histories are formally introduced, in (De Giacomo et al. 2002) De Giacomo et al. introduce epistemically accurate theories and epistemically feasible deterministic programs to theoretically underpin the incremental way of interpreting and executing programs. To still be able to perform projection tasks, they introduce a search operator Σ. Σ carries out the projection while the execution of (exogenous) actions is suppressed.

ccGolog. Grosskreutz and Lakemeyer introduce continuous change into Golog (Grosskreutz and Lakemeyer 2000a; Grosskreutz and Lakemeyer 2003). Based on ideas of (Pinto 1998; Shanahan 1990) with a reified notion of time known from the temporal situation calculus (Pinto and Reiter 1995), fluents can be evaluated on a continuous time scale, similar to (Firby 1994). Each continuous fluent is associated with a function of time. Based on this function and a fluent $start$

which denotes the starting time of an action, fluents can be projected onto an arbitrary point of time in the future. This notion also allows the introduction of a wait-for statement known from RPL (Beetz 2001) into ccGolog which is useful for condition-bounded execution of actions w.r.t. time. Grosskreutz and Lakemeyer (2001) treat the topic of on-line execution vs. off-line projections with the new type of continuous fluents. They propose a system architecture which allows for passive sensing, i.e. sensing is done in the background without actively executing sensing actions. A thorough treatment of ccGolog and the reified time notion can be found in (Grosskreutz 2002; Grosskreutz and Lakemeyer 2003).

pGolog. Grosskreutz and Lakemeyer add the notion of probabilistic programs to Golog in their pGolog dialect (Grosskreutz 2000; Grosskreutz and Lakemeyer 2000b). Programs can be assigned a probability with a statement $prob(p, \sigma_1, \sigma_2)$, where p is a probability and σ_1, σ_2 are pGolog programs. For calculating the probability of a future situation over a probabilistic program they introduce a weighted transition semantics. The semantics is similar to that in ConGolog, though the step semantics is defined as a function mapping pairs of configurations to probabilities ($trans : program \times situation \times program \times situation \rightarrow [0, 1]$). Based on the belief representation of Bacchus et al. (1999) they introduce probabilistic projections into the language. With this extension it is possible to reason if a goal holds in some future situation and with which probability. Further, several initial belief states can be handled with a probability distribution defined over them.

DTGolog. DTGolog (Boutilier et al. 2000) is an approach to integrate decision theory into the Golog framework. In addition to the basic action theory of Golog, an MDP optimization theory is needed. Given a Golog program, DTGolog computes an optimal policy which maximizes the agent's cumulative reward. With the optimization theory DTGolog offers a version of nondeterministic choice of actions and their arguments in an optimized way. DTGolog selects the actions and arguments which lead to the maximal value. The Golog program is interpreted by a predicate $BestDo$ (a version of Do which optimizes the program and calculates the policy). The algorithm used for calculating the policy is a forward search value iteration. Further, DTGolog offers the use of stochastic actions. Stochastic actions in the situation calculus are formalized in the following way. A deterministic action exists which has a number of deterministic outcomes which are chosen by nature. To each outcome a probability is associated, which states the probability with which nature will choose the respective outcome. The resulting policy is a conditional Golog program which offers an optimal action for each of the possible outcomes. In the spirit of Golog, DTGolog also offers to guide the search for an optimal policy by restricting the search space at axiomatizer's needs. Full MDP planning in a finite horizon MDP can be undertaken by using solely nondeterministic action choices. If knowledge about the world exists it can be encoded in the DTGolog program. DTGolog as such is off-line. Soutchanski (2001) proposes an on-line DTGolog interpreter which we discuss in greater detail in Chapter 4.2.1.

GTGolog. GTGolog (Finzi and Lukasiewicz 2004) is an generalization of DTGolog to multi-agent decision-theoretic planning. GT is an abbreviation for "game-theoretic". Formally, multi-player Markov or stochastic games (cf. e.g (von Neumann and Morgenstern 1947; Littman 1994), are solved in their approach. Similar as in DTGolog a program is specified from which the optimal policy is calculated. The optimization seeks at finding a Nash equilibrium for the program by optimizing away nondeterministic choices. Currently, the language is restricted to two-player zero sum Markov games. They give an example in a confined static soccer domain. Finzi and Lukasiewicz (2005) extend their approach to partially observable games. Recently, Farinelli et al. (2007) proposed a version of TeamGolog, a generalization of GTGolog. This dialect focuses on integrating multi-agent decision-theoretic planning and introduces explicit communication and synchronization states.

Applications of Golog

Jenkin et al. (1997) report on interfacing Golog with a Nomad200 and RWI B21 robot. The first realistic large-scale application of Golog was the tour-guide robot in the Deutsches Museum Bonn in the Rhino project (Burgard et al. 1998). Over several days a RWI B21 robot served as museum tour-guide and explained the exhibits. Golog was used here for high-level control. It was connected to the robot system with GOLEX (Hähnel et al. 1998) which interfaced Golog and the rest of the robot control software.

Funge (Funge 1998; Funge 2000) makes use of Golog for modeling animated creatures in a cognitive way. He argues that dealing with uncertainty is important for the animated creatures in order to make it look as realistic as possible, though he does not give a formal approach to it. He uses the possibility of nondeterminism in Golog for his creatures to fill in details in sketch plans based on their background domain knowledge. In an extended example he shows how a merman swims for cover to prevent a shark attack.

Levesque and Pagnucco (2000) report on Legolog, their implementation of Golog on a Lego Mindstorm robot. They connected an IndiGolog interpreter implemented in Prolog to the Lego Mindstorm Robotics Invention System (RIS). The Lego robot is equipped with a micro-controller whose firmware has a C-like API. Further there is a communication protocol with which Prolog and thus IndiGolog are able to communicate with the robot. Actions can be sent to the robot and sensor values can be read. With their implementation, low-cost experiments with cognitive robots can be conducted. This shows the importance of having an embodied physical robot in order to conduct realistic experiments and learn more about the interdependencies between high-level control and a robotic system which may not become apparent in simulations only.

McIlraith and Son (2002) aim at services for the semantic web. As a web agent has to perform complex actions and needs to gather new information, McIlraith and Son chose ConGolog as the specification language for their web agents. The services, encoded as Golog programs, are generic in the sense that different users are able to use them, customizable in the sense that user preferences can be easily integrated, and usable in the sense that agents with different prior knowledge can use them. They extend ConGolog with user constraints by a predicate $Desirable$ to encode user preferences and define self-sufficient programs (Davis 1994) in the situation calculus context. Preferences constrain the search for a solution and self-sufficiency by means of the fluent $KWhether$ ensures that the agent is able to gather all information it needs to execute the program. While McIlraith and Son (2002) make use of ConGolog's nondeterministic choice over actions to generate plans, Fritz and McIlraith (2005) compile qualitative preferences into DTGolog programs. They formally introduce a basic desire language and extend the semantics of DTGolog to handle preference formulas. To ensure the forementioned knowledge gathering process of web agents and to be able to synchronize several web agents working in parallel, they provide a transition semantics for decision-theoretic Golog. In their paper they describe a successful application for booking a business trip considering over 30.000 combinations of flights and hotels, and nearly 1000 queries to the web to gather the needed information.

Pham (2006) describes an interface between DTGolog and the Sony Aibo ERS-7. The interface is based on the framework Tekkotsu (Tira-Thompson 2004). For example, an application that the Aibo is used for is to fulfill navigation tasks for which an optimal policy was calculated.

Soutchanski et al. (2006) discuss the application of DTGolog for the London Ambulance Service case study. The task in this domain is to dispatch an emergency call to an ambulance. An incoming emergency call is taken with all important details. Depending on the location of the request, the request is forwarded to one of three Resource Allocators (there are three as London is separated into three districts). Then, an ambulance is mobilized, either from its home base or on the road, and travels to the scene. At the location the ambulance crew notifies the control center and takes the patients to a nearby hospital if needed. The goal is to minimize the mobilization time to arrive at the scene. A time limit between the incoming request and the arrival at the location is about 14 minutes. The authors show how DTGolog can be applied to this task and show some results in a simulated environment. The results are notable because the state space for this domain is huge.

2.3 Robot Controllers

In this section we briefly overview different control architectures for mobile robots or agents in Section 2.3.1. In Section 2.3.2 we present non-logic-based robot programming languages as a counterpoint to Golog which we discussed in detail before. In Section 2.3.3 we will talk about recent applications (as opposed to those discussed in Section 2.3.1 which mirrors the different paradigms from a more historical viewpoint) and ongoing projects about (cognitive) robotics applications.

2.3.1 Control Architectures

As already noted, the first approach to problem solving on a mobile robot was the STRIPS system (Fikes and Nilson 1971) on the robot Shakey (Nilson 1984). The operation of this robot followed the *sense-plan-act* scheme, where the robot first gathered new information from its sensors, started a planning process to achieve a goal, and finally started to execute the planned sequence of actions. It is the prototype for what was later called the deliberative or hierarchical paradigm. On each of its hierarchical layers sub-goals were to be achieved. The top layer of this architecture, the camera system of Shakey processed information of the world (*sense*). This information was handed over to the STRIPS planner on the middle layer (*plan*). Control commands were passed over to the motors on the bottom layer (*act*).

The drawback of this approach was that due to low computing power a lot of time could pass by between sensing and acting. Thus, the robot could not react appropriately to changes in the environment, as the motor commands were already outdated when they were executed. One proposal to overcome this, was to simply leave out the plan step. In (Brooks 1986; Brooks 1991) Brooks proposed a behavior-based approach. Instead of a vertical hierarchy he proposed a horizontal task composition. This means that several behaviors get the sensor readings as input. The available behaviors like locomotion or collision avoidance had a vertical hierarchy and those behaviors on a higher level subsumed those of a lower level. Therefore, his approach is also called *subsumption architecture*. For the low-level control of a robot this is a good scheme, but from the different available behaviors, goal-directed intelligent acting will merely emerge. In the following times Brooks' thesis that symbolic reasoning hinders the successful development of robots was refuted (e.g. (Gat 1998)).

The solution seems to lie in the middle. In the aftermath, the hybrid paradigm, combining both approaches, was successfully applied. One of the first hybrid architectures is the AuRA architecture (Arkin 1986; Arkin 1987). The deliberative components in his architecture were a mission task planner and a Cartographer. The latter encapsulates all map-building issues. The former is divided into a Mission Planner, Navigator, and Pilot. The Mission Planner deals as a human machine interface, the Navigator plans together with the Cartographer, paths the robot should follow, and the Pilot selects the first sub-task and provides the reactive low-level control (motor modules) with appropriate commands. The planning performed here is an HTN planning approach. Another prominent example is the 3T architecture (Bonasso et al. 1997). It is a three-layered architecture and merges aspects of the ATLANTIS architecture (Gat 1992) and the RAP system (Firby et al. 1995). Three layers are distinguished, a reactive Skill Manager on the bottom level, a deliberative task planner on the top layer, and a sequencer on the middle layer. The top level layer consists of a mission planner which sets the goals to be achieved. These are passed to the sequencer which decomposes the goals making use of HTN techniques. The sequencer calls the low-level behaviors on the bottom layer. This design was deployed successfully for planetary rovers or underwater vehicles at NASA. Simmons et al. proposed the Task Control Architecture (TCA) (Simmons et al. 1997). It is a hybrid approach coming with a task scheduler using PRODIGY (Veloso et al. 1995), a decision-theoretic path planner, and a navigation module deploying POMDP techniques. Interestingly, these are seen as the deliberative layer in (Murphy 2000). The reactive layer consists of an obstacle avoidance scheme based on the curvature-velocity method (Simmons 1996). The robot XAVIER (Simmons et al. 1997) used this architecture in a successful office delivery application.

2.3.2 Non-Logic-Based Robot Programming Languages

PRS-Lite. PRS-Lite (Myers 1996) is a task-level controller for a mobile robot based on the procedural knowledge description of actions by Georgeff and Lansky (1986). The objective was to retain a mixture of goal-directed and reactive behavior in an computational efficient way. PRS-Lite is used as the high-level controller for the robot Flakey from SRI, which uses the Saphira control architecture (Konolige et al. 1997). The control architecture distinguishes between three different control layers, an *effector level*, a *behavior level*, and a *task level*. While the former two deal with controlling sensors and actuators and provide basic behaviors like wall following, the latter deals with the coordination of the control modules present and is implemented using PRS-Lite. A task can be accomplished when all goals of a *goal-set* are satisfied. They distinguish between two different goal modalities, *action* and *sequencing*. For action goals operators exist like tests, execute, and wait-for, which waits for a condition to become true. Further, they can switch on or off intentions and behaviors with *intended*/*unintended* statements which enables or disables the hierarchical decomposition of goals. For sequencing they introduced operators for conditionals, parallel goal execution, and branching. The task procedures are compiled into finite state machines yielding activity schemes. These are launched by instantiating their parameters.

Colbert. Later, PRS-Lite was exchanged by Colbert (Konolige 1997) in the Saphira architecture (Konolige et al. 1997). Colbert is a subset of ANSI C and the semantics of programs is given by finite state automatons (FSA), though the mapping between the FSA and the C program is not formally defined. An input program in Colbert is translated to an FSA which is able to control the robot. Colbert is very similar to PRS-Lite w.r.t. expressiveness. Colbert also makes use of conditionals, wait-for, or goto statements, and allows for hierarchical procedures. Also, concurrent execution of goals is supported.

RAP. The idea behind reactive action packages (RAPs, (Firby 1987; Firby 1994)) is to define hierarchical task nets and expand the nets until only primitive actions remain. Partially specified plans are stored in a plan library. For plan execution those plan skeletons have to be instantiated and details are filled in. The reactive plan interpreter takes a set of tasks or goals and refines each goal hierarchically until primitive actions are reached. This hierarchical refinement is achieved by using RAPs from the plan library. The RAP system is able to handle concurrency and synchronization (wait-for).

Reactive Plan Language, XFRM, and Structured Reactive Controllers. McDermott (1991, McDermott (1992) proposed the *Reactive Plan Language* (RPL). It follows the line of reactive planning and elaborates many ideas of other approaches of reactive planning like RAP described before. RPL is a robot control language coming with rich expressiveness. The language is implemented in LISP, and the semantics of its statement is that of the LISP execution system. *Plans* are understood as programs. RPL features standard imperative constructs like procedures (tasks and sub-tasks), sequences, loops and conditionals, and non-standard constructs like condition- and time-bounded statements like *wait-for* or *wait-time*.[1] These non-standard constructs are based on a notion of *fluents*, which are time-varying variables which refer to sensor data. Fluents are automatically updated when new sensor data arrive. *wait-for* and *wait-time* take a fluent as and argument. RPL offers an account to true concurrency, that is, tasks are executed in parallel (as opposed to interleaved concurrency as for example ConGolog makes use of). This parallelism can be controlled by the programmer by assigned *valves* to a process. Valves are semaphores, and processes competing for valves will not be executed in parallel. This model is also very useful for controlling blocked processes making use of the *wait-for* statement, for example. These processes are suspended until the fluent value becomes true which means that the respective process waits until the associated valve becomes available. With this model of concurrency pure reactive behavior can be modeled. For example, one can assign a process to a fluent with a *with-policy* statement. An example is the behavior policy “whenever the gripper becomes empty, regrasp the object” while other tasks are performed in parallel. Further, RPL introduced a notion for events, where the beginning and the ending of a task can be queried.

It is important to note that the execution system of RPL supports two modes, a *real mode*, where the control programs are executed in reality, and a *projection mode*, where given a simulated time line, programs can be projected into the future. The projections are based on the planning system XFRM (McDermott 1992; Beetz and McDermott 1994). A projected execution is represented by a task network and a time line, where the task network is a tree of the tasks and

[1]Grosskreutz (2002) integrated several features from RPL in his Golog dialect giving them a formal Golog semantics. As we found parts of our Golog language proposal on the work of (Grosskreutz 2002) we will explain some of these statements in Chapter 4 again.

its sub-tasks, and the time line marks start and end points of robot actions. Possible execution scenarios are probabilistically projected, resulting in success and failure cases (see (McDermott 1994) for details on the planning algorithm and the semantics of plans). XFRM offers failure diagnosis for plans. A taxonomy for possible failures exists which might occur during planning. Possible failure cases considered, are for example “object-disappears-after-goal-is-achieved” or “never-achieved-subgoal”. For each failure case a plan revision method is stored which then revises the plan. This methods makes the execution of plans more robust as appropriate behaviors for contingencies are foreseen.

With (Beetz 1999; Beetz 2000; Beetz 2001), Beetz proposes Structured Reactive Controllers (SRC), a control framework for robots which is based on RPL. The idea is to realize robust controllers by *concurrent percept-driven plans*. The system consists of behavior modules, fluents, reactive plans, and the RPL run-time system. Making use of RPL control constructs, like *with-policy*, a controller scheme is defined which allows for plan execution, plan monitoring, and plan revision. Beetz (2001) shows the flexibility of the approach with an application of an office delivery task. The latest extension to RPL is RPLLEARN (Beetz et al. 2004). Following the lines of SRC to formulate an explicit controller framework, Beetz et al. integrate support for learning tasks into the language RPL, that is, the definition of a learning task, how to collect the data needed, which external learning framework should be used, and how to abstract the experiences made during data collection. In an example of a robot navigation tasks they show the usefulness of their approach.

2.3.3 Recent (Cognitive) Robotics and Agent Applications

Cognitive Systems for Cognitive Assistants. (CoSy) is an ongoing integrated EU project of a research consortium consisting of the Royal Institute of Technology (KTH), Sweden, University of Birmingham, the German Research Center for Artificial Intelligence (DFKI), the University of Ljubljana, the University of Freiburg, the University of Paris, and the Technical University Darmstadt. The vision of this project is ”*to construct physically instantiated* [...] *systems that can perceive, understand* [...] *and interact with their environment, and evolve in order to achieve human-like performance in activities requiring context-(situation and task) specific knowledge*” (CoSy 2007). The project comprises of work on architectures, environment representation, object recognition, human-machine interaction and natural language understanding, planning, or action representations (see e.g. (Burgard et al. 2005; Roth et al. 2005; Meier et al. 2006; Kelleher and Kruijff 2006; Kelleher et al. 2006; Kruijff et al. 2006)). One of the key ideas of this project is to integrate work on the different fields of AI and robotics research as well as the cognitive sciences. We refer to the project homepage for more information about the effort and the publication list (CoSy 2007).

The WITAS Project. A prominent example for an application in cognitive robotics is the WITAS project (Doherty et al. 2000). The project’s objective is to control an unmanned aerial vehicle (UVA). The UVA is a modified Yamaha RMAX helicopter which has a power of 16 kW and a maximum take-off weight of 95 kg. It is equipped with three PC104 embedded computers and a CCD camera on a pan/tilt unit. Applications for this UVA are surveillance tasks or to support rescue efforts from the air. An interesting feature in such intelligent UVA applications is that the role of reactive control is predominant. A failing reactive behavior like motion control for a wheeled robot leads to the robot driving on a wrong trajectory or colliding with objects, for an aerial vehicle such a failing control instance has hazardous effects as the helicopter can easily crash. Therefore, Doherty et al. propose a reactive/deliberative architecture where reactive control is of paramount importance. In (Doherty et al. 2004) they introduce a reactive three-layered concentric modular task architecture. The important aspect in this application is to use "*deliberative services in a reactive or contingent manner*" on the one hand, and "*traditional control services in a reactive or contingent manner*" on the other hand. The control modes of the UVA are *take-off, landing via visual navigation, hovering, dynamic path following*, as well as *reactive flight modes for tracking and interception*. From these control modes, it becomes clear that reactive as well as deliberative tasks have to be interwoven. Doherty (2005) describes *DyKnow*, a knowledge processing middle-ware for the helicopter. For tasks like dynamic path following they state that the autonomous system must be able to create qualitative knowledge and objects structures to fulfill such tasks. These representations are a prerequisite for applying planning which "[...] *is an essential functionality in any intelligent agent system*" (Doherty 2005). For planning they make use of the task-based planner TAL (Kvarnström et al. 2000) and a motion planner. TAL is a forward-chaining logical planner where a plan is viewed as a narrative and goals are temporal formulas. Control knowledge can be incorporated by temporal logical formulas which constrain the search. Many publications emanated from this project. We therefore refer to the project's website at (WITAS 2007) for an overview.

The Cognitive Controller. Qureshi et al. (2004) report on a space robotics application. The task is to capture a satellite, transport it to a service bay, perform the service operations, and release the satellite back into orbit. They propose a hybrid system architecture consisting of reactive and deliberative control. The deliberative control guides the reactive one. The authors state that if the satellite is docked, the remaining steps can be performed applying more primitive scripted controllers. The reactive module is implemented as a behavior-based controller and has highest priority to care for safety aspects during operation. There are six different basic behaviors: *search*, *monitor*, *approach*, *align*, *contact*, and *avoid*. Their action selection behavior chooses one of the basic actions that is relevant to achieve the current goal, but it heeds the advice of the deliberative controller. The other low-level components are the *perception center*, which cares for vision processing, and a *memory center* which acts as a world model. For deliberation they make use of a GOLOG planner which plans a sequence of actions from the current state to a predefined goal state. The authors conducted a series of experiments in simulated environments.

Robotic Soccer Applications. One of our application domains for the Golog dialect READYLOG which we propose in Chapter 4 is the robotic soccer domain. A concise discussion of the RoboCup initiative can be found in Chapter 5.2. Here, we want to give some examples for the state of the art in the soccer domain w.r.t. high-level decision making. This overview is not exhaustive, but shall give an impression on how other approaches advance the action selection problem in this domain. One description language which is used by several RoboCup teams is the language XABSL (Lötzsch et al. 2004). XABSL is a XML-based description of hierarchical tasks. The underlying semantics is that of a finite state automaton. Basically, a decision tree is encoded which implements reactive behavior. Murray et al. (2001) make use of so-called statecharts, which are UML-based state automata. The statecharts are implemented in Prolog. Murray et al. (2001) as well as Stolzenburg and Arai (2003) point out that this reactive approach supports the behavior specification in a multi-agent context. A hybrid approach is proposed in (Jensen and Veloso 1998).

Here, simulation league soccer agents also mix reactive and deliberative decision making. Among other things, the authors propose that an agent switches from deliberation to reactive control when an opponent moves too close to the agent. Jensen and Veloso use Prodigy (Veloso et al. 1995), a nonlinear planner, which runs as a central deliberative service and which derives a multi-agent plan for the whole team and then sends each agent its corresponding sub-plan. To make this work, severe restrictions in the expressiveness of the plan language are necessary, like assuming that every action takes the same unit of time. Kok and Vlassis (2006) tackle the problem of multi-agent decision making for simulated soccer agents by so-called coordination graphs. Based on a utility function, each agent selects its next action according to the possible next action of its neighboring agent in this graph. This approach can be seen as a decision tree approach. With the utility function in each situation an appropriate decision tree is selected. As mentioned in the introduction also a range of learning approaches exist. Prominent examples are (Stone 2000; Lauer and Riedmiller 2000; Riedmiller and Merke 2002).

2.4 Discussion

Many different calculi, formalisms and approaches in the field of reasoning about actions and control architectures for mobile robots or agents exist. Each calculus focuses on a different aspect of the reasoning problem. This diversity exists from the early days in these fields. As McCarthy states in (McCarthy 1985) the STRIPS reasoning system was invented because in those days, planning with the situation calculus was too slow for practical applications. An efficient implementation (like with Golog or FLUX) was missing. Many different approaches were published. One could get the impression that the one or the other approach could have been formulated in another logical framework instead of creating a new one. But this, of course, is only partly true. Each formalism has its pros where the aspects in focus can be described in a better, easier, or more appropriate way. There are many papers which enrich one approach with features presented in another one. This simply shows that this field is still under intense investigation. Further, it is remarkable that the approaches from the very beginning (like the situation calculus or STRIPS) are under active research which shows their generality. With the computing power of robots nowadays we are able to implement the visionary ideas from the early days in this field.

With this diversity in the research producing many different approaches, it seems reasonable to justify our approach against the background of other work and thus relating it to the context. As we pointed out in the introduction, the research goals presented in this thesis are to develop robot controllers for highly dynamic domains with real-time constraints. Previous works, mostly theoretical based, gained expressiveness for agent languages. This allows for the modeling of the environment and the behavior of a robot or agent acting in the real world in a more realistic way. The title of this thesis pertains to real-time aspects in the decision making of a robot. This means that, instead of developing an new appealing theoretical concept, we are aiming at showing that existing approaches can — with accordant modifications — be applied to real-time domains. This thesis can be seen in the spirit of gathering common ideas and combining them into one framework rather than creating everything anew. We draw on previous proposals for extensions for the robot programming language Golog and combine them into one dialect.

The first justification is about the question why we have chosen Golog and the situation calculus as basis. The idea of Golog is appealing. The middle ground between planning and programming including temporal projections leaves choices open for modeling the behavior. One can make use of planning where it seems appropriate (and where it is applicable from a computational point of view) and rely on programming techniques where necessary, partly because domain knowledge exists, or because reactive decision making is needed or opportune. The different Golog dialects we integrated in this work makes borrowing from other successful and necessary concepts known, in other also non-logic-based robot programming languages. The language we propose in Chapter 4 melds previous works on Golog which in turn draw on other approaches like reactive planning (e.g. RPL), or decision theory. Concepts of HTN planning, for example, are also (at least partly) contained in the Golog approach with completing hierarchical skeleton plans. Being able to apply these means modeling the agent or robot behavior is one of the strengths of READYLOG. As we will lay out in Chapter 5 where we also show some limitations w.r.t. applicability of the planning capabilities due to the complexity of the application domain, one has nevertheless the chance to successfully tackle complex domains and exhibit intelligent behavior. Again, this is one of the advantages when choosing Golog as a programming language for robot controllers. Another pro of Golog is that with its formal semantics it is well-suited to formulate agent behaviors in a general way (cf. Chapter 7 where we present a general approach to formulate soccer behaviors). Of course, this is also possible with other specification languages basing on a formal semantics. Appealing for us is that this specification can (rather) easily be run on our robot platform (which we present in Chapter 6).

Other concepts like the state representation of the fluent calculus FLUX or the integration of progressing the initial database as shown in $\mathcal{ES}$ are also needed with our approach and had to be modeled in our framework. These papers are very recent, and there is work undertaken to compare, integrate and amalgamate the advantages from both formalisms.

Chapter 3

Mathematical Foundations

In this chapter we will elucidate the mathematical background for this thesis. We start with introducing Markov Decision Processes formally and overviewing standard solution algorithms for decision-theoretic planning in Section 3.1. The sequel of Section 3.1 deals with reinforcement learning as an alternative model to solve Markov Decision Processes under different assumptions and show their relationship to planning techniques. In Section 3.2 we present another Markov model, the hidden Markov model. Unlike in Section 3.1 we do not go into the theory of Hidden Markov models but show Bayes filtering as a solution technique. In particular, we present the Kalman filter and the Particle filter. As Hidden Markov models are used in this thesis for the localization task of a mobile robot, we concentrate on this application for the Particle filter. Section 3.3 presents the formal background of the action formalism we make use of throughout this thesis: the Situation calculus and GOLOG. We present several derivatives of the robot programming language GOLOG which we found our language proposal READYLOG on. Section 3.4 summarizes this chapter.

3.1 Markov Decision Processes

In this section we present *Markov Decision Processes* (MDPs). The class of MDPs is large and different variations of the model exist. To keep things simple, we will focus on fully observable MDPs. For a sound discussion of the different models and optimality criteria we refer to (Puterman 1994; Boutilier et al. 1999).

Markov Decision Processes are a model to describe stochastic dynamic systems. The environment is represented by states. At any time the system can be in one distinct state. The agent makes decisions at certain *stages* or *decision epochs*. One distinguishes between discrete or continuous decision epochs. Here, we deal with discrete decision epochs. At each stage we assume that a state transition occurs due to an action (even if the state remains the same) and therefore equate stage and state transitions. The goal is to devise a conditional plan or *policy* that will maximize the expected benefit of interacting with the environment.

Formally, an MDP is defined by the tuple $M = \langle S, A, T, R \rangle$ where S is a set of states, A is a set of actions, T is a transition function with $T : S \times A \times S \to [0, 1]$, where $T(s_i, a, s_j)$ denotes the probability of the system to take a transition from state s_i to state s_j by applying action a. The function $R : S \rightarrow \mathbb{R}$ is called reward function, assigning a real value to states. The desirability of a particular state is expressed with the reward assigned to it by R. The set of actions and the set of states can be of infinite cardinality. Here, we only regard finite state and action sets. The transition function must be defined in such a way that it satisfies the Markov property: the probability of the next state depends on the current state and action only, and not on any of the previous states. It means that

$$T(s,a,s') = Pr(S^{t+1} = s'|A^t = a, S^t = s_t, ..., S^1 = s_1) = Pr(S^{t+1} = s'|A^t = a, S^t = s),$$

where t represents the stage of the system. A policy π for an MDP assigns actions from A to states in S. It can be seen as a conditional plan which for each state gives advice as to which action to perform. It is called *stationary* if the advice for performing an action in a particular state is independent of the current stage of the system, otherwise it is called *non-stationary*. Thus, a stationary policy $\pi : S \rightarrow A$ can be represented by a fixed matrix, where for non-stationary policy $\pi : S \times \{1, \ldots, T\} \rightarrow A$ one such matrix for each stage $1, \ldots, T$ exists. One can distinguish between MDPs having an *infinite* number of decision epochs and those having a *finite horizon*. MDP models which have a definite terminal stage, i.e. go into an absorbing state, but where the number of stages to reach this absorbing state is not known in advance are called *indefinite-horizon* MDPs.

Further, one mainly distinguishes between *fully observable* and *partial observable* problems (although there exist models which are *non-observable*). In the former, by performing an action the agent cannot predict the state which it will attain, but it can sense the state it is in with certainty. In the latter case of partial observability, it has a probability distribution about a possible set of states it might be in. Here, we deal with fully observable MDPs.

For a finite-horizon problem the value of a policy is the sum of the expected rewards the agent receives following the policy π:

$$V(s) = \mathrm{E}_\pi \left[\sum_{t=0}^{T} R(s_t | s_0 = s) \right], \quad V(s) = \mathrm{E}_\pi \left[\gamma \cdot \sum_{t=0}^{\infty} R(s_t | s_0 = s) \right]. \tag{3.1}$$

where t denotes the stage the system is in and γ, $0 \leq \gamma < 1$, is a factor which discounts the rewards over the time for the finite and infinite-horizon problem, respectively. The discount factor γ assures that the cumulated expected reward does not grow unboundedly. In the following we regard discounted problems.

Now, the agent is interested in finding the optimal course of actions. The expectation in Eq. 3.1 can be rewritten as

$$V^\pi(s_i) = R(s_i) + \gamma \sum_{s_j \in S} T(s_i, \pi(s_i), s_j) \cdot V^\pi(s_j) \tag{3.2}$$

Eq. 3.2 is referred to as the Bellman equation (Bellman 1957). Each policy π establishes a system of $|S|$ linear equations with $|S|$ unknown variables. The equation states that the value of state s_i under policy π is the sum of the immediate reward received in state s_i plus the discounted expected reward of the successor states of s_i. Howard (1960) showed the existance of a unique optimal value function V^* resulting from any optimal policy π^*. The optimal policy can be derived from the equation system defined by Eq. 3.2 by choosing greedily the action a in state s which maximizes the expected value of all successor states that are reachable by action a.

In the next section we show three algorithms which derive the optimal policy. We want to note again that there are several different MDP models (as well as semi-MDPs or POMDP models which are also related). Several other optimality criteria exist. For an in-depth discussion of all the subtleties of the different (PO)MDP models we refer to (Puterman 1994) and to (Boutilier et al. 1999) for a thorough overview.

3.1.1 Decision-theoretic Planning

Decision-theoretic (DT) planning is the process of deriving an optimal policy for an MDP. The policy provides for each system state the optimal policy the agent should take in order to maximize the value defined in terms of immediate rewards for a certain state and for the actions taken so far. DT planning requires that the transition model of the system is known. In the next section we will show some methods to derive a policy for the case that the model is not known in advance. Before that, we present the *value iteration*, and *policy iteration* algorithms, and a search based method to derive optimal policies.

Value Iteration

Bellman (1957) shows that the expected value of a policy can be computed by

$$V_t^{\pi}(s_i) = R(s_i) + \sum_{s_j \in S} T(s_i, \pi(s_i, t), s_j) \cdot V_{t-1}^{\pi}(s_j).$$

$V_t^{\pi}(s)$ is the *t-stage-to-go value function*. At decision epoch t the value for a state is dependent on the immediate reward given in the respective state plus the value of the states which lead the agent to the current state weighted by the transition probability. A policy is called optimal iff $\forall \pi', s \in S : V_T^{\pi}(s) \geq V_T^{\pi'}(s)$ at the final decision epoch. Thus, there is the following relationship between the optimal value function at stage t and the optimal value function at the previous stage:

$$V_t^{*}(s_i) = R(s_i) + \max_{a \in A} \sum_{s_j \in S} T(s_i, a, s_j) \cdot V_{t-1}^{*}(s_j). \tag{3.3}$$

Bellman's principle of optimality (Bellman 1957) which roughly states that every decision of an optimal policy is optimal disregarding the initial state and the initial decisions, builds the basis for dynamic programming techniques. Generalizing Eq 3.3 to discounted infinite-horizon problems Howard (1960) showed that there is always a stationary policy for such problems. The optimal value function satisfies

$$V^{*}(s_i) = R(s_i) + \max_{a \in A} \gamma \sum_{s_j \in S} T(s_i, a, s_j) \cdot V^{*}(s_j).$$

These equations form the basis for value iteration. As the equations generate a definite equation systems, it can be solved iteratively (dynamic programming). The policy results from taking the optimal action at each state. In the case of the infinite-horizon problem it has been shown that V_t converges linearly to the true value function V^{*}.

Policy Iteration

The idea of the *policy iteration algorithm* (Howard 1960) for infinite-horizon problems is slightly different. Instead of improving the value function and deriving the policy from the optimal choices, it directly modifies the policies. The algorithm has two steps: *policy evaluation* and *policy improvement*. In the former step the value of the policy $V^{\pi_i}(s)$ is calculated for each $s \in S$ based on the policy π_i. The algorithm starts with an arbitrary policy π_0. In the policy improvement step the action a^* has to be found which maximizes the state-action function

$$Q_{i+1}(s_i, a) = R(s_i) + \gamma \sum_{s_j \in S} T(s_i, a, s_j) \cdot V^{\pi_i}(s_j). \qquad (3.4)$$

If $Q_{i+1}(s_i, a^*) > V^{\pi_i}(s_i)$ then action a^* is selected, i.e. $\pi_{i+1}(s_i) = a^*$, otherwise $\pi_{i+1}(s_i) = \pi_i(s_i)$. Although the complexity of policy iteration is higher, in practice it turned out that it converges in fewer iterations than the value iteration approach.

Decision Tree Search

Boutilier et al. (1999) describe a decision tree search algorithm to find an optimal policy for a finite-horizon MDP $M = \langle S, A, T, R \rangle$ and horizon H. An example of a decision tree is depicted in Figure 3.1. Decision Tree Search works as follows. From an initial situation $s_0 \in S$ the possible actions $a \in A$ are "executed". As the actions are stochastic they lead to successor states with certain probabilities. In the example action a_1 leads with probability p_1 to s_1 and with p_2 to s_2, analogously for the action a_2 with resulting states s_3 and s_4. In the successor states, actions are again executed, leading to a tree alternating action and situation nodes. Edges between action and situation nodes are annotated with the probability to reach the particular successor state by executing an action. The edges from situations to actions are annotated with values V. The value at the leaves $V(s, H) = R(s)$, i.e. leaves of the tree, are assigned the reward of the particular state. The value of parent states calculates as the sum of the values of its children weighted by the probability to reach that specific child. An example is given in Figure 3.1 for $V_2 = p_3 \cdot V_3 + p_4 \cdot V_4$. At each state the maximum value of all its sub-trees is taken. The action which leads to the maximal value is the optimal action on that stage and will be part of the optimal behavior policy.

One problem with this approach is that the whole tree has to be expanded to be able to compute the optimal policy. Further, the branching factor for realistic problems can be immense and can become a problem. For infinite-horizon problems a tree has to be constructed which is deep enough to ensure convergence of the value function. Another method which is search-based is Real-time dynamic programming (Barto et al. 1995). A partial decision tree is expanded based on a search heuristic. If the agent needs to take a decision the values are calculated. This method approximates the value function, and the deeper the tree the more accurate the value function is approximated. For more reading about search-based methods we refer to (Boutilier et al. 1999).

3.1.2 Reinforcement Learning and the Link to DT Planning

A different approach to find an optimal behavior policy for Markov Decision Processes is Reinforcement Learning (RL). The general difference to DT planning is that the transition model is

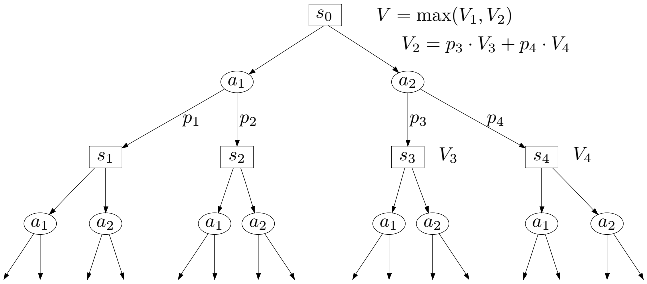


Figure 3.1: Decision tree search (from (Boutilier et al. 1999))

not given for RL problem instances. Instead, the agent tries out actions in its environments and observes the outcomes of its acting in terms of a reinforcement signal (reward).

Many different solution methods for approximating the value function exits, like Monte Carlo approaches, Temporal Difference Learning, or learning with function approximation (cf. e.g. (Kaelbling et al. 1996; Sutton and Barto 1998)). Here, as an exemplary approach, we show Q-learning, a temporal difference technique. Watkins (1989) proposed Q-learning. The optimal state-action function is approximated in terms of performing actions in the environment, observing the feed-back, and minimizing the approximation error from previous updates. It can be seen as a model-free policy improvement (cf. Eq. 3.4). The new state-action value is updated according to a learning rate α, i.e. $Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot Q(s_{t+1}, a_{t+1})$, and thus $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$. Then, the basic update formula for *off-policy one-step Q-Learning* follows as

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t))$$

The state-action function is approximated by following a behavior policy π. The value for the state-action pair (a_t, s_t) is then updated by taking the reward for the observed situation and the Q-value for the best action to be performed afterwards into account. The difference to DT methods is that only one sample ($Q(s_{t+1}, a)$) is taken to update the current state-action value, instead of the whole state update used in DT planning ($\sum T(s, a, s')V(s)$). α is the learning rate influencing to which degree new experiences contribute to a state-action value, γ is a discount factor. It is guaranteed that Q-learning converges to the optimal state-action function Q^* in the limit, i.e. each state is visited infinitely often. This update scheme is called off-policy as the behavior policy is not directly changed. In contrast to this the SARSA algorithm is an *on-policy* method. The update scheme looks very similar:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)).$$

The difference is that the policy is directly changed when new experiences are incorporated. Summarizing, instead of calculating a behavior policy with a given transition model as in DT planning,

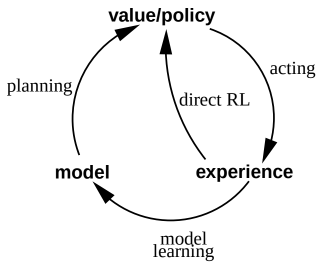


Figure 3.2: Relationships between learning, planning and acting (from (Sutton and Barto 1998))

RL methods make observations by executing actions in a trial-and-error fashion to approximate the optimal value (state-action) function.

In (Sutton and Barto 1998, Chap. 9) Sutton and Barto give a good overview how decision-theoretic planning, reinforcement learning and model learning are related to each other. It is given in Figure 3.2. Having a policy and acting in the environments leads to making experiences. Direct RL methods use these experiences to update the behavior policy. This can be seen as policy improvement in model-free policy iteration methods. If these experiences are used to increase the transition model one speaks of model learning. Having a model (learned or not) to generate a behavior policy is the key to DT planning.

3.2 Bayes Filtering for Robot Localization

Another approach to deal with uncertainty is that of Bayes filtering. The general idea is to estimate a hidden state variable x of a dynamical system from observations that can be made. The underlying model for Bayes filters are Hidden Markov Models (HMM), which in this case can be seen as simple Bayesian nets. These models enjoy the Markov property as it is assumed that the particular state x_t at a time t of the system only depends on x_{t-1}, that is, $p(x_t|x_0 \ldots, x_{t-1}) = p(x_t|x_{t-1})$. Further, the observation made at time t is only dependent on the state x_t and is conditionally independent of all other states, i.e. $p(o_t|x_0, \ldots, x_t) = p(o_t|x_t)$.

The key idea of Bayes filtering is to estimate a probability density over the state space conditioned on the observation made. It is an estimate as the true state is unknown (hidden). The belief $Bel(x_t)$ of the system being in state x_t is defined by the posterior

$$Bel(x_t) = P(x_t|o_1, \ldots, o_t)$$

A Bayes filter consists of the following two steps:

1. *Prediction step:* At each time step a prediction of the state is made:

$$Bel^-(x_t) = \int p(x_t|x_{t-1}) \cdot Bel(x_{t-1}) dx_{t-1}.$$

$p(x_t|x_{t-1})$ describes the system dynamics and is also referred to as the motion model.

2. *Correction step:* Each time a new observation is made it is used to correct the predicted belief

$$Bel(x_t) = \eta \cdot p(o_t|x_t) \cdot Bel^-(x_t).$$

The likelihood of an observation o_t assumed that the system is in state x_t is expressed by the perceptual model $p(o_t|x_t)$. η is a normalizing constant which ensures that the posterior over the entire state space sums up to one.

Bayes filter have a wide application range. In our context it is used mainly for localizing a robot in its environment (Chapter 5). The observations then can be seen as data coming from the robot's sensors. The state variable $x_t = (x, y, \theta)^\mathsf{T}$ represents the pose of the robot, with (x, y) being the position, and θ the orientation of the robot. When localizing a robot with a Bayes filter, $Bel(x_0)$ is initialized with prior knowledge about the position. Often, the initial pose in unknown and $Bel(x_0)$ is initialized with a uniform probability distribution, as all positions are equally likely. To localize a robot with no information about the initial position, is called *global localization*. If the robot has a high belief about its position the process of localization is called *local localization* or *position tracking*.

In the following we briefly introduce two Bayes filters which we will use in Chapter 6.

3.2.1 Kalman Filter

The Kalman filter (Kalman 1960) is a linear Bayes filter with Gaussian distribution. The state variable $x_t \in \mathbb{R}^n$ and the observations $o_t \in \mathbb{R}^m$ are formulated by the equations

$$x_t = Ax_{t-1} + Bu_{t-1} + w_{t-1} \quad \text{and} \quad o_t = Hx_t + v_t.$$

The $n \times n$ matrix A is the transition model stating the relationship of the state variable x between two time steps. With the term $B \cdot u$ an additional control input can be given. w and v are random variables and are adding noise to x and o. They are assumed to be independent of each other and normally distributed with $p(w) \sim N(0, Q)$ and $p(v) \sim N(0, R)$. Q is the process noise covariance and R the measurement noise covariance matrix.[1] The matrix H relates observable components of the measurement and the state.

Let $\mu_t^- \in \mathbb{R}^n$ be the prior, and $\mu_t \in \mathbb{R}^n$ the posterior state estimate at step t, i.e. $\mu_t^- = \mathrm{E}\left[x_t^-\right]$ and $\mu_t = \mathrm{E}\left[x_t\right]$. The prior and posterior state estimation error can be defined as $e_t^- \equiv x_t - \mu_t^-$ and $e_t \equiv x_t - \mu_t$. The respective error covariances are $\Sigma_t^- = \mathrm{E}\left[e_t^- e_t^{-\mathsf{T}}\right]$ and $\Sigma_t = \mathrm{E}\left[e_t e_t^\mathsf{T}\right]$. The posterior state estimate μ_t is calculated by $\mu_t = \mu_t^- K(o_t - H\mu_t^-)$, where the difference $o_t - H\mu_t^-$ is called innovation and reflects the difference between the predicted measurement $H\mu_t^-$ and the actual measurement o_t. K is called Kalman gain and is calculated by

$$K_t = \Sigma_t^- H^\mathsf{T}(H\Sigma_t^- H^\mathsf{T} + R)^{-1} = \frac{\Sigma_t^- H^\mathsf{T}}{H\Sigma_t^- H^\mathsf{T} + R}.$$

The Kalman gain is needed to calculate the posterior error covariance from the prior one (in the correction step of the filter below). The idea is to minimize the mean-square error of a state given a

[1]Recall that the covariance of a random variables X $\Sigma(X) = \mathrm{E}\left[(X - \mathrm{E}\left[X\right])\right]$.

state prediction of the previous step and new observations, i.e. $\mathrm{E}\left[(x_t - \mu_t)^2\right]$ is to be minimized. This is equivalent to minimizing the posterior error covariance Σ_t; the above formula is the result of minimizing Σ_t. The derivations are given for example in (Thrun et al. 2005). Given these equations, one can establish the both steps of a Bayes filter mathematically.

1. *Prediction step:* In the prediction step the a posteriori state estimate together with its error covariance is calculated:

$$\begin{aligned}\mu_t^- &= A\mu_{t-1} + Bu_{t-1}\\ \Sigma_t^- &= A\Sigma_{t-1}A^\mathsf{T} + Q\end{aligned}$$

   where Q is the process noise covariance. It means that the a posteriori prediction error plus the noise which is inherent to the whole process forms the next a priori error.

2. *Correction step:* The just calculated a priori error is minimized and with it the new state estimate is established. Finally, we correct the a posteriori error needed for the next iteration of the filter

$$\begin{aligned}K_t &= \Sigma_t^- H^\mathsf{T}(H\Sigma_t^- H^\mathsf{T} + R)^{-1}\\ \mu_t &= \mu_t^- + K_t(o_t - H\mu_t^-)\\ \Sigma_t &= (I - K_tH)\Sigma_t^-\end{aligned}$$

In the notation given in the introduction the respective density functions are $p(x_t|x_{t-1}) = N(Ax_{t-1}, Q)$, $p(o_t|x_t) = N(Hx_t, R)$, and $p(x_{t-1}|o_0, \ldots, o_{t-1}) = N(\mu_{t-1}, \Sigma_{t-1})$, with N denoting the normal distributed probability density function..

The Kalman filter has a variety of applications. Each linear stochastic process with Gaussian noise where the state of the system is only indirectly given through observations can be modeled by Kalman filters. Kalman filters are optimal estimators. Extensions to deal with non-linear models also exist (e.g. (Maybeck 1990)). Regarding the localization of a mobile robot one can say that Kalman filters can only be used for position tracking. The reason is that the belief of the position is represented by a uni-modal Gaussian and the filter is thus unable to represent multiple hypotheses. An extension regarding this issue is *Multiple Hypothesis Tracking* (MHT) (Bar-Shalom and Li 1995) where for each single hypothesis a separate Kalman filter is used.

3.2.2 Particle Filter

Next, we briefly describe an approach to robot localization using Monte Carlo methods. For the localization problem, these methods extend the so-called *Markov localization*. Markov localization uses a grid-based state representation of the pose $\langle x, y, \theta \rangle$ of the robot. The environment map is represented as a 3D occupancy grid (Moravec and Elfes 1985) where cells which are occupied have a high occupancy value and free cells are assigned the occupancy value zero. The localization of the robot is pursued relative to this map.

The Markov localization method is a recursive Bayes filter. The initial belief about the robot's pose is represented by a uniform distribution over the whole state space for the case of global localization, or by a Gaussian with a certain small mean and variance around a given initial position. The recursive update equations are:

1. *Prediction step:*

$$\begin{aligned} Bel^-(x_t) &= \int p(x_t|x_{t-1}, a_{t-1}, o_{t-1}, \dots, a_0, o_0) p(x_{t-1}|a_{t-1}, \dots, o_0) dx_{t-1} \\ &= \int p(x_t|x_{t-1}, a_{t-1}) p(x_{t-1}|a_{t-1}, \dots, o_0) dx_{t-1} \\ &= \int p(x_t|x_{t-1}, a_{t-1}) Bel(x_{t-1}) dx_{t-1} \end{aligned}$$

   The derivation follows again from the Markov assumption. The o_i are as above the observations the robots made, i.e. the sensor inputs, the a_i are odometry updates. These are sensor values for the wheel encoder of the robot and represent the actions (drive commands) of the robot.

2. *Correction step:* In the correction step the Markov assumption is applied again. It is assumed that the measurements only depend on the current position of the robot.

$$\begin{aligned} Bel(x_t) &= \eta \cdot p(o_t|x_t, a_{t-1}, o_{t-1}, \dots, a_0, o_0) \cdot Bel^-(x_t) \\ &= \eta \cdot p(o_t|x_t) \cdot Bel^-(x_t) \end{aligned}$$

The belief distribution is updated by a discretized approximation of $Bel(x_t)$ according to x_t. Each grid cell in the occupancy grid must be touched for the update. Though extensions which only update the most important regions at the current time step exist, there are more efficient methods for approximating the belief distribution. Using sampling techniques for this approximation is far more efficient. In the following we present the particle filter following the notation given in (Thrun et al. 2000). $Bel(x)$ is represented by set of weighted samples

$$Bel(x) \approx \{x^{(i)}, w^{(i)}\}_{i=1,\dots,m},$$

where $x^{(i)}$ is one out of m samples of the random variable x and represents one hypothesis, a pose in the case of localization. The importance factor $w^{(i)}$ assigns a weight to this hypothesis, the sum of all weights is to be one, i.e. $\sum_{i=0}^{m} w^{(i)} = 1$. One Monte Carlo method is the SIS (sequential importance sampling) filter. It is a bootstrap filter which means that from an initial guess after several iterations the filter converges towards the real distribution. It is also called bootstrap filtering, particle filter, or condensation algorithm. The recursive update is performed by the following steps.

1. *Prediction step:* A number of samples $x_{t-1}^{(i)}$ from $Bel(x_{t-1})$ and samples $x_t^{(i)}$ according to $p(x_t|x_{t-1}, a_{t-1})$ with $i = 1, \dots, m$ are drawn. The tuples $\langle x_t^{(i)}, x_{t-1}^{(i)} \rangle$ are distributed according to $Bel^-(x_t)$. $Bel^-(x_t)$ restated for a sample pair $\langle x_t^{(i)}, x_{t-1}^{(i)} \rangle$ is then

$$Bel^-(x_t^{(i)}) = p(x_t^{(i)}|x_{t-1}^{(i)}, a_{t-1}) Bel(x_{t-1}^{(i)})$$

The particles $\langle x_t^{(i)}, x_{t-1}^{(i)} \rangle$ do not reflect the target distribution $Bel(x)$ at time t. The error is corrected in the next step.

2. *Correction step:* Each sample is now weighted according to $\hat{w}^{(i)} = p(o_t|x_t^{(i)})$. The distribution $p(o_t|x_t)$ is the sensor model. In the localization context, it is easy to calculate the probability of making a sensor reading o_t given the pose x_t. A commonly used method to estimate the probability is to use ray-tracing methods on the environment map given. For each single proximity measurement a probability is calculated with aid of the ray-tracing method. The whole distribution for j single measurements is calculated by $p(o|x) = \prod_j p(o_j|x)$. $\hat{w}$ represents the target distribution except for a factor:

$$w_t^{(i)} = \frac{\eta p(o_t|x_t^{(i)}) p(x_t^{(i)}|x_{t-1}^{(i)}, a_{t-1}) Bel(x_{t-1}^{(i)})}{p(x_t^{(i)}|x_{t-1}^{(i)}, a_{t-1}) Bel(x_{t-1}^{(i)})} = \frac{\eta p(o_t|t_t^{(i)}) Bel^-(x_t^{(i)})}{Bel^-(x_t^{(i)})} = \eta \hat{w}_t^{(i)}.$$

This means that $w = \eta \cdot \hat{w}$, the weights $\hat{w}$ gained from the sensor model and the approximation w of the target distribution are proportional to each other. Normalizing $\hat{w}$ yield the correctly distributed values to approximate $Bel(x_t)$

$$Bel(x_t^{(i)}) = \eta \cdot \hat{w}^{(i)} \cdot Bel^-(x_t^{(i)}).$$

The sampling process is repeated m times resulting in a set of m weighted samples $x_t^{(i)}$. Initially, the $w_0^{(i)}$ are set to $1/m$.

Monte Carlo methods for localizing a robot are very successful. Many different approaches and refinements exist for the class of Monte Carlo localization algorithms, like dual sampling, or mixture MCL. For a thorough discussion we refer to (Thrun 2006; Montemerlo et al. 2006).

3.3 Reasoning about Action and Change

The formal foundation of READYLOG is the Situation calculus (McCarthy 1963; Levesque et al. 1998). We will introduce the Situation calculus in Section 3.3.1 and the language GOLOG in Section 3.3.2. Section 3.3.3 introduces the on-line transition semantics and presents an account to integrate sensing actions. In Section 3.3.4 we focus on extensions to deal with continuous change and probabilistic programs. Finally, in Section 3.3.5 we present off-line decision-theoretic GOLOG.

Notational Conventions

In the following logical formulas we use the standard logical notations and several abbreviations. We use the logical connectives "$\neg$" (negation), "$\wedge$" (conjunction), "$\vee$" (disjunction), "$\supset$" (implication), "$\equiv$" (equivalence), "$\forall$" (universal quantification), and "$\exists$" (existential quantification). The scope of quantifiers is indicated with parentheses. Alternatively, we use the "dot" notation which indicates that the quantifier preceding the dot has maximum scope. So, $\forall x.P(x) \supset Q(x)$ stands for $(\forall x)[P(x) \supset Q(x)]$. We often omit parentheses assuming that the connectives have the following precedence ordered from high to low: $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$. $\neg P \wedge Q \vee$

$R \wedge S \supset T \equiv U \wedge V$ stands for $(((\neg P \wedge Q) \vee (R \wedge S)) \supset T) \equiv (U \wedge V)$. We use the abbreviation $\exists x_1, \ldots, x_n.P(x_1, \ldots, x_n)$ for the expression $\exists x_1.\exists x_2. \ldots .\exists x_n.P(x_1, \ldots x_n)$, and $\forall x_1, \ldots, x_n.P(x_1, \ldots, x_n)$ for $\forall x_1.\forall x_2. \ldots .\forall x_n.P(x_1, \ldots, x_n)$. Sometimes, we use $\vec{x}$ for the sequence of pairwise different variables $x_1, \ldots, x_n$ and $\vec{x} = \vec{y}$ for $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$. Moreover, we use the convention that all "free" variables in sentences are implicitly universally quantified. Therefore $\exists y.y > x$ stands for $\forall x.\exists y.y > x$.

3.3.1 Situation Calculus

The situation calculus is a second order language with equality which allows for reasoning about actions and their effects. The world evolves from an initial situation due to primitive actions. Possible world histories are represented by sequences of actions. The situation calculus distinguishes three different sorts: *actions*, *situations*, and domain dependent *objects*.

A special binary function symbol $do : action \times situation \rightarrow situation$ exists, with $do(a, s)$ denoting the situation which arises after performing action a in the situation s. The constant S_0 denotes the initial situation, i.e. the situation where no actions have yet occurred. We abbreviate the expression $do(a_n, \cdots do(a_1, S_0) \cdots)$ with $do([a_1, \ldots, a_n], S_0)$.

The state the world is in is characterized by functions and relations with a situation as their last argument. They are called *functional* and *relational fluents*, resp. As an example consider the position of a robot navigating in an office environment. One aspect of the world state is the robot's location $robotLoc(s)$. Suppose the robot is in an office with room number 6214 in the initial situation S_0. The robot now travels to office 6215. The position of the robot then changes to $robotLoc(do(travel, S_0)) = 6215$. $travel$ denotes the robot's action of traveling from office 6214 to 6215. The situation the world is in is described by $s_1 = do(travel, S_0)$. The value of the functional fluent $robotLoc(s_1)$ equals 6215.

The third sort of the situation calculus is the sort *action*. Actions are characterized by unique names. For each action one has to specify a *precondition axiom* stating under which conditions it is possible to perform the respective action and an *effect axiom* formulating how the action changes the world in terms of the specified fluents. An action precondition axiom has the form

$$Poss(a(\vec{x}), s) \equiv \Phi(\vec{x}, s) \tag{3.5}$$

where the binary predicate $Poss : action \times situation$ denotes when an action can be executed, and $\vec{x}$ stands for the arguments of action a. For our travel action the precondition axiom may be $Poss(travel, s) \equiv robotLoc(s) = 6214$. After having specified when it is physically possible to perform an action it remains to state how the respective action changes the world. One has to specify negative and positive effects in terms of the relational fluent F:

$$\varphi^+(\vec{x}, s) \quad \supset \quad F(\vec{x}, do(a, s)) \tag{3.6}$$

$$\varphi^-(\vec{x}, s) \quad \supset \quad \neg F(\vec{x}, do(a, s)). \tag{3.7}$$

The effect axiom for a functional fluents f is

$$\varphi_f(\vec{x}, y, a, s) \supset f(\vec{x}, do(a, s)) = y.$$

Describing the positive and the negative effect says nothing about those effects which do not change the fluent. With a completeness assumption all effects have to be modeled. Consider a robot lifting a box. The effect of this action is that the robot is holding the box afterwards. But there is no axiom which defines that the color of the box is not changed by the lift action. The problem of describing the non-effects of an action is referred to as the *frame problem*. The number of frame axioms is very large. For relational fluents there exist in the order of $2 \cdot \mathcal{A} \cdot \mathcal{F}$ frame axioms, where $\mathcal{A}$ is the number of actions, and $\mathcal{F}$ the number of relational fluents. McCarthy and Hayes (1969) were the first to mention this problem.

Reiter's Solution to the Frame Problem

A solution to the problem was proposed by Reiter (1991). The solution is based on the idea to collect all effect axioms about a given fluent. With a completeness assumption that the collected effects are the only ways to change the fluent's value a *successor state axiom* is derived for the fluent which yields a solution to the frame problem. The positive and negative effects for a fluent F are grouped together:

$$\begin{aligned} \Phi_F^{(1)} &\supset F(\vec{x}, do(a,s)) \\ &\vdots \\ \Phi_F^{(k)} &\supset F(\vec{x}, do(a,s)) \end{aligned}$$

which can be logical equivalently written as

$$\left(\Phi_F^{(1)} \vee \cdots \vee \Phi_F^{(k)}\right) \supset F(\vec{x}, do(a,s)).$$

Similarly, for the negative effects. This leads to the so-called positive and negative normal form (Eq. 3.6 and 3.7). Now it is assumed that these axioms characterize all the conditions under which action a causes F to become true or false, resp., in the successor situation. Thus, the two sentences Eq. 3.6 and 3.7 describe the causal laws for F. Now, explanation closure is applied:

$$\begin{aligned} F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a,s)) &\supset \varphi_F^-(\vec{x}, a, s) \\ \neg F(\vec{x}, s) \wedge F(\vec{x}, do(a,s)) &\supset \varphi_F^+(\vec{x}, a, s). \end{aligned}$$

The idea behind these axioms is that if the truth value of F changes from false to true from situation s to situation $do(a,s)$ then $\varphi_F^+(\vec{x}, a, s)$ must have been true. Similarly, for the second axiom. Further, a unique names assumption for actions is needed. It states that for distinct action names A and B it holds that $A(\vec{x}) \neq B(\vec{x})$ and $A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}$. Reiter shows that if a first-order theory that entails $\neg\exists \vec{x}, a, s. \varphi_F^+(\vec{x}, a, s) \wedge \varphi_F^-(\vec{x}, a, s)$, then T entails that together with the explanation closure axioms the normal form axioms for fluent F are logically equivalent to

$$F(\vec{x}, do(a,s)) \equiv \varphi_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\varphi_F^-(\vec{x}, a, s). \tag{3.8}$$

The above formula is called *successor state axiom for the relational fluent F*. The successor state axiom for the functional fluent f has the form (Reiter 2001):

$$\begin{aligned} & f(\vec{x}, do(a,s)) = y \equiv \\ & \quad \varphi_f(\vec{x}, y, do(a,s)) \vee f(\vec{x}, s) = y \wedge \neg\exists y'. \varphi_f(\vec{x}, y', a, s) \end{aligned} \tag{3.9}$$

The background theory must entail the consistency property

$$\neg\exists\vec{x}, y, y', a, s.\varphi_f(\vec{x}, y, a, s) \wedge \varphi_f(\vec{x}, y', a, s) \wedge y \neq y'. \tag{3.10}$$

The number of $\mathcal{F}$ successor state axiom together with $\mathcal{A}$ action precondition axiom plus the unique names axioms is far less than the $2 \cdot \mathcal{F} \cdot \mathcal{A}$ explicit frame axioms that would be needed otherwise.

Basic Action Theories

The background theory is a set of sentences $\mathcal{D}$ consisting of

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0},$$

where

- Σ is the set of foundational axioms for situations. The set Σ consists of the following foundational axioms for situations:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2 \tag{3.11}$$
$$\forall P.P(S_0) \wedge \forall a, s.(P(s) \supset P(do(a, s))) \supset \forall s.P(s) \tag{3.12}$$
$$\neg s \sqsupset S_0 \tag{3.13}$$
$$s \sqsupset do(a, s') \equiv s \sqsupseteq s' \text{ with } s \sqsupseteq s' \text{ is an abbreviation for } s \sqsupset s' \vee s = s' \tag{3.14}$$

  The intuition about these axioms is that situations are axiomatized as finite sequences of actions and that certain sequences are subsequences of others. Axiom 3.11 is a unique names axiom for situations. Situation terms should thus be seen as action histories rather than as world states. The second axiom 3.12 is a situation closure axiom limiting the sort *situation* to the smallest set containing S_0 which is closed under application of the function do. Axiom 3.13 and 3.14 define an ordering on situations.

- $\mathcal{D}_{ssa}$ is a set of successor state axioms for functional and relational fluents, one for each fluent as given in Eq. 3.8 for relational fluents and in Eq. 3.9 for functional fluents (together with the consistency property Eq. 3.10).

- $\mathcal{D}_{ap}$ is a set of action precondition axioms, one for each action. The set $\mathcal{D}_{ap}$ is the set of precondition axioms of the form $Poss(a(\vec{x}), s)$ as described above.

- $\mathcal{D}_{una}$ is the set of unique names axioms for all actions.

- $\mathcal{D}_{S_0}$ is a set of first order sentences that are uniform in S_0 and describe the fluent values in the initial situation.[2]

[2]Sentences uniform in s are sentences which do not quantify about situations, nor mention $Poss$ or $\sqsupset$.

Regression

To address the so-called projection problem, i.e. determining if a sentence holds for some future situations, a regression mechanism is used in the situation calculus. Basically, if one wants to prove that a sentence W is entailed by the basic action theory and W mentions a relational fluent $F(\vec{t}, do(a,\sigma))$ (with $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$ being F's successor state axiom) one determines a logically equivalent formula W' by substituting $\Phi_F(\vec{t}, \alpha, \sigma)$ for $F(\vec{x}, do(a,\sigma))$. By this substitution, the complex situation term $do(\alpha,\sigma)$ has been eliminated from W'. By successively applying regression until W' is uniform in S_0 the problem of proving if W is entailed in some complex situation reduces to a theorem proving task in the initial database. We use Reiter's definition for regressing relational fluents. A more general definition including functional fluents is given in (Pirri and Reiter 1999).

Definition 1 (Regressable Formulas) *A formula W of $\mathcal{L}_{sitcalc}$[3] is regressable iff*

1. *Each term of sort situation mentioned by W has the syntactic form $do([\alpha_1, \ldots, \alpha_n], S_0)$ for some $n \geq 0$, where $\alpha_1, \ldots, \alpha_n$ are terms of sort action.*
2. *For each atom of the form $Poss(\alpha, \sigma)$ mentioned by W, α has the form $A(t_1, \ldots, t_n)$ for some n-ary action function symbol A of $\mathcal{L}_{sitcalc}$.*
3. *W does not quantify over situations.*
4. *W does not mention the predicate symbol $\sqsubset$, nor does it mention any equality atom $\sigma = \sigma'$ for terms σ, σ' of sort situation.*

Definition 2 (The Regression Operator) *Let W be a regressable formula, $\mathcal{D}$ a basic action theory with $\mathcal{D}'_{ssa} \subset \mathcal{D}_{ssa}$ the set of successor state axioms for relational fluents and $\mathcal{D}_{ap}$ the set of precondition axioms. $\vec{t}$ is a tuple of terms, α is a term of sort action, σ a term of sort situation. The regression operator $\mathcal{R}$ is inductively defined over the structure of W.*

1. *If W is a situation-independent atom or a relational fluent atom of the form $F(\tau, S_0)$, then*

   $$\mathcal{R}[W] = W.$$

2. *If W is a regressable $Poss$ atom of the form $Poss(A(\vec{t}), \sigma)$, there must exist a precondition axiom of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ in $\mathcal{D}_{ap}$. Then*

   $$\mathcal{R}[W] = \mathcal{R}[\Pi_A|^{\vec{x},s}_{\vec{t},\sigma}].$$

3. *W is a relational fluent atom of the form $F(\vec{t}, do(\alpha, \sigma))$. Then, in $\mathcal{D}'_{ssa}$ there exists an axiom $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$. Then*

   $$\mathcal{R}[W] = \mathcal{R}[\Phi_F|^{\vec{x},s}_{\vec{t},\sigma}].$$

[3]We leave out the formal definition of $\mathcal{L}_{sitcalc}$ in this thesis referring to (Pirri and Reiter 1999; Reiter 2001).

4. *For formulas, regression is defined inductively as*

$$\mathcal{R}[\neg W] \equiv \neg\mathcal{R}[W]$$
$$\mathcal{R}[W_1 \wedge W_2] \equiv \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$$
$$\mathcal{R}[\exists v.W] \equiv \exists v.\mathcal{R}[W].$$

A complete definition of the regression operator including functional fluents is given in (Pirri and Reiter 1999). Pirri and Reiter (1999) showed that

$$\mathcal{D} \models W \text{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[W],$$

with W a regressable sentence of $\mathcal{L}_{sitcalc}$ and $\mathcal{D}$ a basic action theory. This means that the evaluation of regressable sentences can be reduced to a theorem proving task in the initial theory $\mathcal{D}_{S_0}$ together with unique names axioms for actions. No successor state or precondition axioms are needed for this task, no foundational axioms Σ are needed. With this result at hand one can compute whether a ground situation is executable with the regression operator. The executable situations are defined as

$$executable(s) \overset{def}{=} \forall a, s^*.do(a, s^*) \sqsubseteq s \supset Poss(a, s^*).$$

It can be shown that

$$\Sigma \models \forall a_1, \ldots, a_n.executable(do[a_1, \ldots, a_n], S_0) \equiv \bigwedge_{i=1}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}], S_0).$$

Further it was shown that

$$\begin{aligned}&\mathcal{D} \models executable(do([a_1, \ldots, a_n], S_0))\\ &\text{iff}\\ &\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \bigwedge_{i=1}^{n} \mathcal{R}[Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_0)).\end{aligned} \tag{3.15}$$

3.3.2 Golog

The high-level programming language GOLOG ("alGOL in LOGic") (Levesque et al. 1997) is based on the situation calculus. As planning is known to be computationally very demanding and impractical for deriving complex behaviors with hundreds of actions, GOLOG finds a compromise between planning and programming. The robot or agent is equipped with a situation calculus background theory. The programmer can specify the behavior just like in ordinary imperative programming languages but also has the possibility to project actions into the future. The amount of planning (projection) used is in the hand of the programmer. With this, one has a powerful language for specifying the behaviors of a cognitive robot or agent. The semantics of GOLOG is defined by situation calculus macros. Programs are expanded to situation calculus formulas with the predicate Do. If the theory entails the behavior program δ

$$\mathcal{D} \models (\exists\delta, s).Do(\delta, S_0, s) \wedge Goals(s),$$

it is proven that program δ leads to the goal situation s starting from S_0. As a side effect of the constructive proof one yields an executable GOLOG program.

The Semantics of Golog

The semantics is defined as follows.

1. **Primitive Action:** $Do(a, s, s') \overset{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s).$
   For a primitive action it is checked if it is possible to execute it in the current situation (denoted by the predicate $Poss$). Note that $a[s]$ denotes the action term with its situation argument restored. The successor situation is then $do(a[s], s)$.

2. **Test action:** $Do(\varphi?, s, s') \overset{def}{=} \varphi[s] \wedge s' = s.$
   Similar to primitive actions in the formula $\varphi[s]$ all situation arguments are restored. A test does not change the situation.

3. **Sequence:** $Do([\delta_1; \delta_2], s, s') \overset{def}{=} \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s').$

4. **Nondeterministic choice of actions:** $Do((\delta_1|\delta_2), s, s') \overset{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$

5. **Nondeterministic choice of action argument:** $Do((\pi x)\delta(x), s, s') \overset{def}{=} \exists x.Do(\delta(x), s, s').$

6. **Nondeterministic iteration:**

$$\begin{aligned} Do(\delta^*, s, s') \quad \overset{def}{=} \quad & \forall P.(\forall s_1.P(s_1, s_1) \wedge \\ & \forall s_1, s_2, s_3.(P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3))) \supset P(s, s') \end{aligned}$$

7. **Conditionals:** $Do(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif}, s, s') \overset{def}{=} Do([\varphi?; \delta_1|\neg\varphi?; \delta_2], s, s').$

8. **Loops:** $Do(\textbf{while } \varphi \textbf{ do } \delta \textbf{ endwhile}, s, s') \overset{def}{=} Do(((\varphi?; \delta)^*; \varphi?), s, s').$

9. **Recursive Procedures:** To define recursive procedures one needs to define an auxiliary macro definition. For any predicate P of arity $n + 2$ define $Do(P(t_1, \ldots, t_n), s, s') \overset{def}{=} P(t_1[s], \ldots, t_n[s], s, s')$. Expressions of the form $P(t_1, \ldots, t_n)$ serve as procedure calls. Then, one has to define the situation calculus semantics for programs involving recursive procedures making use of a standard block-structured programming style. A program will have the form

$$\textbf{proc } P_1(\vec{\vartheta}_1)\ \delta_1 \textbf{ endproc}; \cdots; \textbf{proc } P_n(\vec{\vartheta}_n)\ \delta_n \textbf{ endproc}; \delta_0,$$

   where $P_1, \ldots, P_n$ with formal parameters $\vec{\vartheta}_i$ and procedure bodies $\delta_1, \ldots, \delta_n$ are procedures. The result of evaluating a program is defined as

$$\begin{aligned} & Do(\{\textbf{proc } P_1(\vec{\vartheta}_1)\delta_1 \textbf{ endproc}; \cdots; \textbf{proc } P_n(\vec{\vartheta}_n)\delta_n \textbf{ endproc}\}; \delta_0, s, s') \overset{def}{=} \\ & \quad \forall P_1, \ldots, P_n. \left[\bigwedge_{i=1}^{n} (\forall s_1, s_2.\vec{\vartheta}_i).Do(\delta_i, s_1, s_2) \supset P_i(\vec{\vartheta}_i, s_1, s_2)\right] \supset Do(\delta_0, s_1, s_2) \end{aligned}$$

The Famous Elevator Example

The elevator example is famous in that it is usually a standard example for GOLOG and was often adduced. We follow this by presenting the elevator program from (Reiter 2001).

The available actions are $up(n)$, $down(n)$, $turnoff(n)$, $open$, and $close$, which moves the elevator up and down to floor n, turns off the call button at floor n, and opens or closes the door, resp. The fluents needed are $currentFloor(s) = n$, and $on(n, s)$, giving the current floor and stating whether the call button on floor n is turned on or not. The action preconditions are (1) $Poss(up(n), s) \equiv currentFloor(s) < n$, (2) $Poss(down(n), s) \equiv currentFloor(s) > n$, (3) $Poss(open, s) \equiv true$, (4) $Poss(close, s) \equiv true$, and (5) $Poss(turnoff(n), s) \equiv on(n, s)$. The successor state axioms for the fluents $currentFloor$ and on are

$$\begin{aligned}
&currentFloor(do(a, s)) = m \equiv a = up(m) \lor a = down(m) \lor \\
&\quad currentFloor(s) = m \land \neg\exists n.a = up(n) \land \neg\exists n.a = down(n) \\
&on(m, do(a, s)) \equiv on(m, s) \land a \neq turnoff(m).
\end{aligned}$$

As an abbreviation $nextFloor(n, s)$ is defined as

$$\begin{aligned}
&nextFloor(n, s) \overset{def}{=} on(n, s) \land \\
&\quad \forall m.on(m, s) \supset |m - currentFloor(s)| \geq |n - currentFloor(s)|,
\end{aligned}$$

which is the nearest floor to the currently served floor. Further we need the procedures defined below. The initial situation is given as $\mathcal{D}_{S_0} = \{currentFloor(S_0) = 4, on(b, S_0) \equiv b = 3 \lor b = 5\}$.

```
proc $serves(n)$
  $goFloor(n)$; $turnoff(n)$; $open$; $close$ endproc
proc $goFloor(n)$
  $(currentFloor = n)?$ | $up(n)$ | $down(n)$ endproc
proc $park(n)$
  if $currentFloor = 0$ then $open$ else $down(0)$; $open$ endif
endproc
proc $control$
  [while $\exists n.on(n)$ do $serveAFloor$ endwhile]; $park$
endproc
```

To run the program $control$ means to do theorem proving for the entailment

$$Axioms \models \exists s.Do(control, S_0, s).$$

A successful "execution" returns the following binding for s:

$$\begin{aligned}
&s = do([down(3), turnoff(3), open, \\
&\quad close, up(5), turnoff(5), open, close, down(0), open], S_0)).
\end{aligned}$$

A Golog Interpreter in Prolog

Next, we briefly address the Prolog implementation of GOLOG. We show the Toronto implementation of GOLOG from (Reiter 2001). The semantics is defined by a set of clauses `do/3` which directly map the *Do* predicate to Prolog code:

$$\mathtt{do}(\mathtt{E}_1 \ : \ \mathtt{E} : 2, \mathtt{S}, \mathtt{S}_1) :- \mathtt{do}(\mathtt{E}_1, \mathtt{S}, \mathtt{S}_2), \mathtt{do}(\mathtt{E}_2, \mathtt{S}_2, \mathtt{S}_1).$$

$$\mathtt{do}(?(\mathtt{P}), \mathtt{S}, \mathtt{S}) :- \mathtt{holds}(\mathtt{P}, \mathtt{S}).$$

$$\mathtt{do}(\mathtt{E}_1 \# \mathtt{E}_2, \mathtt{S}, \mathtt{S}_1) :- \mathtt{do}(\mathtt{E}_1, \mathtt{S}, \mathtt{S}_1); \mathtt{do}(\mathtt{E}_2, \mathtt{S},_1).$$

$$\mathtt{do}(\mathtt{if}(\mathtt{P}, \mathtt{E}_1, \mathtt{E}_2), \mathtt{S}, \mathtt{S}_1) :- \mathtt{do}(?(\mathtt{P}) \ : \ \mathtt{E}_1 \# ?(-\mathtt{P}) \ : \mathtt{E}_2, \mathtt{S}, \mathtt{S}_1).$$

$$\mathtt{do}(\mathtt{star}(\mathtt{E}), \mathtt{S}, \mathtt{S}_1) :- \mathtt{S}_1 = \mathtt{S}; \mathtt{do}(\mathtt{E} \ : \ \mathtt{star}(\mathtt{E}), \mathtt{S}, \mathtt{S}_1).$$

$$\mathtt{do}(\mathtt{while}(\mathtt{P}, \mathtt{E}), \mathtt{S}, \mathtt{S}_1) :- \mathtt{do}(\mathtt{star}(?(\mathtt{P}) \ : \ \mathtt{E}) \ : ?(-\mathtt{P}), \mathtt{S}, \mathtt{S}_1).$$

$$\mathtt{do}(\mathtt{pi}(\mathtt{V}, \mathtt{E}), \mathtt{S}, \mathtt{S}_1) :- \mathtt{sub}(\mathtt{V}, \_, \mathtt{E}, \mathtt{E}_1), \mathtt{do}(\mathtt{E}_1, \mathtt{S}, \mathtt{S}_1).$$

$$\mathtt{do}(\mathtt{E}, \mathtt{S}, \mathtt{S}_1) :- \mathtt{proc}(\mathtt{E}, \mathtt{E}_1), \mathtt{do}(\mathtt{E}_1, \mathtt{S}, \mathtt{S}_1).$$

$$\mathtt{do}(\mathtt{E}, \mathtt{S}, \mathtt{do}(\mathtt{E}, \mathtt{S})) :- \mathtt{primitive\_action}(\mathtt{E}), \mathtt{poss}(\mathtt{E}, \mathtt{S}).$$

Further, a predicate `sub/4` is needed for term substitutions. These are for instance fluent values which have to be instantiated:

$$\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{T}_1, \mathtt{T}_2) :- \mathtt{var}(\mathtt{T}_1), \mathtt{T}_2 = \mathtt{T}_1.$$

$$\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{T}_1, \mathtt{T}_2) :- \mathtt{not}\ \mathtt{var}(\mathtt{T}_1), \mathtt{T}_1 = \mathtt{X}_1, \mathtt{T}_2 = \mathtt{X}_2.$$

$$\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{T}_1, \mathtt{T}_2) :- \mathtt{not}\ \mathtt{T}_1 = \mathtt{X}_1, \mathtt{T}_1 =.. \ [\mathrm{F} \mid \mathrm{L}_1], \mathtt{sub\_list}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{L}_1, \mathtt{L}_2), \mathtt{T}_2 =.. \ [\mathrm{F} \mid \mathrm{L}_2].$$

$$\mathtt{sub\_list}(\mathtt{X}_1, \mathtt{X}_2, [\,], [\,]).$$

$$\mathtt{sub\_list}(\mathtt{X}_1, \mathtt{X}_2, [\mathrm{T}_1 \mid \mathrm{L}_1], [\mathrm{T}_2 \mid \mathrm{L}_2]) :- \mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{T}_1, \mathtt{T}_2), \mathtt{sub\_list}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{L}_1, \mathtt{L}_2).$$

The `do` clause for a test mentions the predicate `holds/2`. It is needed to evaluate logical formulas. It is defined by the following set of clauses:

$$\mathtt{holds}(\mathtt{P}\ \&\ \mathtt{Q}, \mathtt{S}) :- \mathtt{holds}(\mathtt{P}, \mathtt{S}), \mathtt{holds}(\mathtt{Q}, \mathtt{S}).$$

$$\mathtt{holds}(\mathtt{P}\ \mathrm{v}\ \mathtt{Q}, \mathtt{S}) :- \mathtt{holds}(\mathtt{P}, \mathtt{S}); \mathtt{holds}(\mathtt{Q}, \mathtt{S}).$$

$$\vdots$$

$$\mathtt{holds}(-(-\mathtt{P}), \mathtt{S}) :- \mathtt{holds}(\mathtt{P}, \mathtt{S})$$

$$\vdots$$

$$\mathtt{holds}(-\mathtt{all}(\mathtt{V}, \mathtt{P}), \mathtt{S}) :- \mathtt{holds}(\mathtt{some}(\mathtt{V}, -\mathtt{P}), \mathtt{S}).$$

$$\mathtt{holds}(-\mathtt{some}(\mathtt{V}, \mathtt{P}), \mathtt{S}) :- \mathtt{not}\ \mathtt{holds}(\mathtt{some}(\mathtt{V}, \mathtt{P}), \mathtt{S}).$$

$$\mathtt{holds}(-\mathtt{P}, \mathtt{S}) : -\mathtt{isAtom}(\mathtt{P}), \mathtt{not}\ \mathtt{holds}(\mathtt{P}, \mathtt{S}).$$

$$\mathtt{holds}(\mathtt{all}(\mathtt{V}, \mathtt{P}), \mathtt{S}) :- \mathtt{holds}(-\mathtt{some}(\mathtt{V}, -\mathtt{P}), \mathtt{S}).$$

$$\mathtt{holds}(\mathtt{some}(\mathtt{V}, \mathtt{P}), \mathtt{S}) :- \mathtt{sub}(\mathtt{V}, \_, \mathtt{P}, \mathtt{P}_1), \mathtt{holds}(\mathtt{P}_1, \mathtt{S}).$$

Finally, for (test) actions one needs a predicate which restores the situation argument of an action term, as well as a check if a formula is atomic.

```
holds(A, S) :- restoreSitArg(A, S, F), F;
    not restoreSitArg(A, S, F)isAtom(A), A.
isAtom(A) :- not(A = -W; A = (W_1 & W_2); A = (W_1  =>  W_2);
    A = (W_1  <=>  W_2); A = (W_1 v W_2); A = some(X, W); A = all(X, W)).
restoreSitArg(poss(A), S, poss(A, S)).
```

These clauses suffice to implement a GOLOG interpreter in Prolog. Reiter thoroughly examines the assumption for a correct Prolog implementation. One problem is how Prolog works on negations. With the "negation as finite failure" strategy, correctness depends on the ordering of Prolog clauses. Negated facts should always ordered past non-negated ones.

This gives an impression of how compact the implementation of the agent programming language GOLOG is. But many important features to control robots in dynamic domains are missing in the language. Throughout the next sections we will present several extensions to the language GOLOG, without regarding their implementation. In Chapter 4 we come back to the implementation issues of READYLOG. The implementation here shall be seen as a reference implementation to the improvements made over recent years.

3.3.3 The Transition Semantics and Guarded Action Theories

While GOLOG is well-suited to reason about actions and their effects, it has the drawback that a program has to be evaluated up to the end before the first action can be performed. It might be that the world changed between plan generation and plan execution so that the plan is not appropriate or is invalid. GOLOG interpreters based on the so-called evaluation semantics as shown above are therefore called off-line interpreters. Another problem that arises is that the knowledge of the agent must be complete. There are no actions which allow the agent to gather new knowledge during operation, i.e. it is not able to deal with incomplete knowledge.[4]

To overcome the problems with the evaluation semantics inherent in GOLOG, De Giacomo et al. (2000) proposed an incremental interpreter with CONGOLOG. The program is interpreted in a step-by-step fashion where a transition relation defines the transformation from one step to another. In this so-called transition semantics a program is interpreted from one configuration $\langle\sigma, s\rangle$, a program σ in a situation s, to another configuration $\langle\delta, s'\rangle$ which results after executing the first action of σ, where δ is the remaining program and s' the situation resulting of the execution of the first action of σ. The one-step transition function $Trans$ defines the successor configuration for each program construct, and thus the semantics of the language construct. Clearly, one needs termination conditions for programs. Therefore, there is the existence of another predicate $Final$

[4]Note that some extensions exists which integrate sensor actions into the evaluation semantics like (Lakemeyer 1999). But they do not overcome the problem of GOLOG being off-line.

which defines the final configuration, i.e. those configuration where no further transition can be made and thus, the program terminates.

The definition of $Trans$ is given below.

$$
\begin{aligned}
&Trans(nil, s, \delta, s') \equiv \mathit{false}\\
&Trans(\alpha, s, \delta, s') \equiv Poss(a[s], s) \wedge \delta = nil \wedge s' = do(a, s)\\
&Trans(\varphi?, s, \delta, s') \equiv \varphi[s] \wedge \delta = nil \wedge s' = s\\
&Trans([\delta_1; \delta_2], s, \delta, s') \equiv Final(\delta_1, s) \wedge\\
&\qquad Trans(\delta_2, s, \delta, s') \vee \exists \delta'.\delta = (\delta'; \delta_2 \wedge Trans(\delta_1, s, \delta', s)\\
&Trans(\delta_1 \mid \delta_2, s, \delta, s') \equiv Trans(\delta_1, s, \delta, s') \vee Trans(\delta_2, s, \delta, s')\\
&Trans(\pi v.\delta, s, \delta, s') \equiv \exists x.Trans(\delta|_x^v, s, \delta, s')\\
&Trans(\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endif}, s, \delta, s') \equiv\\
&\qquad \varphi[s] \wedge Trans(\delta_1, s, \delta, s') \vee \neg\varphi[s] \wedge Trans(\delta_2, s, \delta, s')\\
&Trans(\mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1\ \mathbf{endwhile}, s, \delta, s') \equiv\\
&\qquad \varphi[s] \wedge \exists \delta'.\delta = (\delta'; \mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1\ \mathbf{endif}) \wedge Trans(\delta, s, \delta', s')\\
&Trans(\sigma_1 \gg \sigma_2, s, \delta, s') \equiv\\
&\qquad \exists \gamma.\delta = (\gamma \gg \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee\\
&\qquad \exists \gamma.\delta = (\sigma_1 \gg \gamma) \wedge Trans(\sigma_2, s, \gamma, s') \wedge \forall \gamma', s''.\neg Trans(\sigma_1, s, \gamma', s'')\\
&Trans(\sigma_1 \,||\, \sigma_2, s, \delta, s') \equiv\\
&\qquad \exists \gamma.\delta = (\gamma \,||\, \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee \exists \gamma.\delta(\sigma_1 \,||\, \gamma) \wedge Trans(\sigma_2, s, \gamma, s')
\end{aligned}
$$

nil is the empty program, which can be seen as a termination action. Therefore, a transition is not possible, but the empty program fulfills the final condition. As in GOLOG for a *primitive action* a it is tested if it is possible. The successor configuration is $\langle nil, do(a, s)\rangle$ and the respective final predicate does not hold. The semantics of the other predicates is similar to that of GOLOG. Note that conditionals and loops in CONGOLOG are synchronized in contrast to GOLOG in the sense that no other action between the test and the execution of the first action can happen.

The transition function allows for a semantics of concurrency. The first construct $\sigma_1 \gg \sigma_2$ has the meaning that σ_1 is preferred over σ_2; σ_1 is executed whenever possible. Only if there does not exist any successor transition for σ_1, σ_2 is executed. Both, σ_1 and σ_2 have to be final for the prioritized execution to become final. The other form of concurrency is the statement $\sigma_1 \,||\, \sigma_2$. None of the two programs are preferred over the other. The actions are truly interleaved.

$$
\begin{aligned}
&Final(nil, s) \equiv \mathit{true}\\
&Final(\alpha, s) \equiv \mathit{false}\\
&Final(\varphi?, s) \equiv \mathit{false}\\
&Final([\delta_1; \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)\\
&Final([\delta_1 \mid \delta_2], s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)\\
&Final(\pi v.\delta, s) \equiv \exists x.Final(\delta|_x^v, s)\\
&Final(\delta^*, s) \equiv \mathit{true}\\
&Final(\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endif}, s) \equiv\\
&\qquad \varphi[s] \wedge Final(\delta_1, s) \vee \neg\varphi[s] \wedge Final(\delta_2,)s
\end{aligned}
$$

$$Final(\textbf{while } \varphi \textbf{ do } \sigma \textbf{ endwhile}, s) \equiv \varphi[s] \wedge Final(\sigma, s)$$
$$Final(\sigma_1 \gg \sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$$
$$Final(\sigma_1 \,||\, \sigma_2) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$$

The predicate $Do(\sigma, s, s')$ defines the situation s' which is reachable from s by executing program σ leading to a final configuration. It is defined in terms of the transitive closure $Trans^*$ over $Trans$:

$$Do(\sigma, s, s') \equiv \exists\delta.Trans^*(\sigma, s, \delta, s') \wedge Final(\delta, s').$$

Thus, s' is the situation where program σ reaches a final configuration by successively taking transitions with $Trans$.

$$Trans^*(\sigma, s, \delta, s') \equiv \forall T(\ldots \supset T(\sigma, s, \delta, s'))$$

where the ellipsis stands for the universal closure of the conjunction of the following formulas:

$$T(\sigma, s, \sigma, s)$$
$$Trans(\sigma, s, \sigma^*, s^*) \wedge T(\sigma^*, s^*, \delta, s') \supset T(\sigma, s, \delta, s').$$

Unlike GOLOG, where the language constructs are defined as abbreviations of situation calculus formulas, the transition semantics demands the encoding of CONGOLOG programs as first-order terms. A thorough treatment of this can be found in (De Giacomo et al. 2000).

De Giacomo et al. proposed INDIGOLOG, a GOLOG dialect whose semantics is also based on the transition semantics (De Giacomo and Levesque 1999; De Giacomo et al. 2001). It accounts for on-line execution and off-line projections and introduces a notion of sensing actions with so-called guarded action theories.

The set of successor state axioms $\mathcal{D}_{ssa}$ of in the basic action theory is replaced by two sets of of axioms:

- $\mathcal{D}_{GSSA}$, a set of guarded successor state axioms of the form $\alpha(\vec{x}, a, s) \supset (F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s))$, where α is a fluent formula called guard of the axiom, F is a relational fluent and γ is a fluent formula,

- $\mathcal{D}_{GSFA}$, a set of guarded sensed fluents axioms of the form $\beta(\vec{x}, s) \supset (F(\vec{x}, do(a, s)) \equiv \rho(\vec{x}, s)$.

To show the difference between $GSSA$ and $GSFA$ we reprint an example from (De Giacomo et al. 2001). It is from the elevator domain which we introduced above. An example for a $GSSA$ formula is

$$ControllerOn(s) \supset (Light(x, do(a, s)) \equiv (a = on(x) \vee Light(x, s) \wedge a \neq \mathit{off}(x)))$$

The antecedent is a guard for evaluating the successor state axiom. It corresponds to $\alpha(\vec{x}, a, s)$ above. An example for a $GSFA$ guard is the following axiom:

$$Floor(x, s) \supset (Light(x, s) \equiv sensor\_floor(s) > 50).$$

The idea is that if the sensor value for the fluent $sensor\_floor$ is greater than 50 one can conclude that the light at floor x is on.[5]

To keep track of the sensed values, De Giacomo et al. introduce the notion of *histories*, a sequence of $(\vec{\mu_0} \cdot (a_1, \vec{\mu_1}) \cdot \cdots \cdot (A_n, \vec{\mu_n}))$ where A_i are ground action terms and the $\vec{\mu_i} = (\mu_{i1}, \ldots, \mu_{im})$ is a vector of sensed values. μ_{ij} can be seen as the value of the jth sensor after the ith action. Let σ be history. Then the ground action term $end[\sigma]$ is defined as $end[\vec{\mu_0}] = S_0$ and $end[\sigma \cdot (A, \vec{\mu})] = do(A, end[\sigma])$. A ground sensor formula $Sensed[\sigma]$ is defined as $\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{m} h_j(end[\sigma_i]) = \mu_{ij}$, where σ_i is a sub-history up to action $i, \sigma_i = (\vec{\mu_0}) \cdots (A_i, \vec{\mu_i})$, and h_j is the jth sensor function.

An on-line execution of a program δ_0 is a sequence $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$ such that for $i = 0, \ldots, n-1$ $Axioms \cup Sensed[\sigma_i] \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}])$, with $\sigma_{i+1} = \sigma_i$, if $end[\sigma_{i+1}] = end[\sigma_i]$, and otherwise $\sigma_{i+1} = \sigma_i \cdot (a, \vec{\mu})$, if $end[\sigma_{i+1}] = do(a, end[\sigma_i])$ and $\vec{\mu}$ is the sensor result after a. The on-line execution is successful if $Axioms \cup Sensed[\sigma_n] \models Final(\delta_n, end[\sigma_n])$. It is unsuccessful if $Axioms \cup Sensed[\sigma_n] \not\models Final(\delta_n, end[\sigma_n])$ and there exists no δ_{n+1} and s such that $Axioms \cup Sensed[\sigma_n] \models Trans(\delta_n, end[\sigma_n], \delta_{n+1}, s)$. $Axioms$ denote the background theory $\mathcal{D}$ plus the axioms defining $Trans$ and $Final$. A history σ is executable iff either $\sigma = \vec{\mu_0}$, or $\sigma = \sigma' \cdot (A, \vec{\mu})$, σ' is an executable history, and $\mathcal{D} \cup Sensed[\sigma'] \models Poss(A, end[\sigma'])$.

With the notion of executability for programs with active sensing we have a means to decide whether a program mentioning active sensing actions can be prevented from being blocked due to a missing sensor value with performing projections. We will not elaborate on this matter referring to (De Giacomo et al. 2001). Important for our interpreter is that the implementation of these action histories with sensing values which we also rely on are proved to be correct w.r.t. guarded action theories (De Giacomo et al. 2001).

3.3.4 ccGolog and pGolog

Grosskreutz (2002) proposed the extensions CCGOLOG and PGOLOG. CCGOLOG deals with continuous change, an extension to overcome the limitation of GOLOG that the world only evolves from situation to situation and, especially for projections, continuous processes cannot be modeled. PGOLOG extends GOLOG with a notion of probability. With this it is possible to assign a probability to a program and with that reason with uncertainty. So, it can be projected if a formula φ holds with a certain probability.

Continuous Change

In CCGOLOG a new type of fluent is introduced, so-called continuous fluents. As a motivating example consider a navigation task of a robot. A fluent $robotLoc$ denotes the location of the robot. When an action $startGo(v)$ with v the speed of the robot is initiated the position of the robot at each point in time can be characterized by the function $pos(t) = x_0 + v \cdot (t - t_0)$. Fluents, whose values can be characterized by a function of time, can be modeled as continuous fluents.

[5]We make use of these kind of guards in our implementation of READYLOG. In the implementation (Chapter 5) these guards are encoded as side conditions to effect axioms.

He therefore introduces a time line into CCGOLOG and a new sort *t-function*, which are functions of time. New axioms val are added to the action theory defining the value of a *t-function*:

$$
\begin{aligned}
&val(constant(x_0, \dots, x_n), t) = \langle x_0, \dots, x_n \rangle \\
&val(linear(x_0, \dots, x_n, v_0, \dots, v_n, t_0), t) = \langle x'_0, \dots, x'_n \rangle \equiv \\
&\qquad x'_0 = x_0 + v_0 \cdot (t - t_0) \land \dots \land x'_n = x_n + v_n \cdot (t - t_0)
\end{aligned}
$$

With these axioms and the time line it is possible to define condition-bounded actions. The new action type is the $waitFor(\tau)$ action. The argument τ is called $t\text{-}form$, a Boolean combination of closed atomic formulas of the form $(F\ op\ r)$ with F being a continuous fluent, $op \in \{<, =\}$, and r a real number. $\tau[s, t]$ denotes the evaluation of a $t\text{-}form$ in situation s at time t. Every occurrence of a continuous fluent F in the formula τ is replaced by $val(F(s), t)$. The fluent formula is replaced by its *t-function* on s and t. The idea of the $waitFor$ action is that as soon as the condition τ holds the action terminates. To evaluate this point in time the notion of the least time point $ltp(\tau, s, t)$ is needed. It is an abbreviation for the formula

$$
\tau[s, t] \land t \geq start(s) \land \forall t'.start(s) \leq t' < t \supset \neg\tau[s, t'], \tag{3.16}
$$

which is the earliest point in time where after the start of s τ becomes true. With notion at hand, one can define the precondition axiom for $waitFor(\tau)$:

$$
Poss(waitFor(\tau), s) \equiv \exists t.ltp(\tau, s, t).
$$

The successor state axiom for the fluent $start$ mentioned in Eq. 3.16 is modeled as

$$
\begin{aligned}
&Poss(a, s) \supset (start(do(a, s)) = t \equiv \\
&\qquad \exists \tau.a = waitFor(\tau) \land ltp(\tau, s, t) \lor \forall \tau.a \neq waitFor(\tau) \land t = start(s))
\end{aligned}
$$

With the notion of the least time point the semantics of concurrently executing programs changes to

$$
\begin{aligned}
&Trans((\sigma_1 \,||\, \sigma_2), s, \delta, s') \equiv \\
&\qquad \neg Final(\sigma_1, s) \land \neg Final(\sigma_2, s) \land \\
&\qquad (\exists \delta_1.Trans(\sigma_1, s, \delta_1, s') \land \delta = (\delta_1 \,||\, \sigma_2) \land \\
&\qquad\quad (\forall \delta_2, s_2.Trans(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)) \lor \\
&\qquad (\exists \delta_2.Trans(\sigma_2, s, \delta_2, s') \land \delta = (\sigma_1, \delta_2) \land \\
&\qquad\quad (\forall \delta_1, s_1.Trans(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1))) \\
&Final(\sigma_1 \,||\, \sigma_2, s) \equiv Final(\sigma_1, s) \lor Final(\sigma_2, s)
\end{aligned}
$$

If a transition from $\langle \sigma_1, s \rangle$ to $\langle \delta_1, s' \rangle$ exists and no transition of σ_2 with an earlier start point, σ_1 is executed. Clearly, the statement is final if σ_1 or σ_2 is final. We omit the semantic definitions of the other language constructs of CCGOLOG here. We will come back to them in Chapter 4 as part of the language READYLOG.

Further, Grosskreutz (2002) proposes a control architecture where the high-level GOLOG controller settles commands via a special fluent $register$ by performing a special $send$ action. By this, the low-level task is invoked and runs asynchronously to the high-level controller. When the low-level task is accomplished, it signals this by means of a $reply$ message. For example, when the robot should navigate to a certain position, a $startGo$ action is initiated. The high-level controller can go on with other tasks, while the low-level system of the robot will control the navigation task. When the robot reaches its destination it signals the completion of the navigation task. This is modeled as an exogenous action.

The successor state axiom for the action register is

$$\begin{aligned}&Poss(a,s) \supset [reg(id,do(a,s)) = val \equiv\\ &\quad a = send(id,val) \vee a = reply(id,val) \vee\\ &\quad\quad reg(id,s) = val \wedge \neg\exists v.(a = send(id,v) \vee a = reply(id,v))]\end{aligned}$$

This allows the high-level controller more flexibility. As a natural further step this architecture demands that complex tasks have so-called low-level models. The high-level controller just invokes low-level tasks which are conducted by the robot control system. Nevertheless, the high-level controller must have an idea what the low-level system does. For projection tasks, where no action is executed the high-level controller needs a model of the low-level tasks in order to predict the results appropriately. A projection over a low-level module is defined as

$$Proj(s,\sigma,ll_{model},s') \overset{def}{=} Do(s,withPol(ll_{model},\sigma,s')),$$

where $withPol(ll_{model},\sigma)$ is an abbreviation for $((ll_{model}; false?) \,||\, \sigma)$. σ is a high-level program, and ll_{model} is a program which models the effects of σ on the low-level system. The program and the low-level task are executed concurrently. The $withPol$ statement guarantees that both, σ and ll_{model}, become final.

Two special actions $setOnline$ and $clipOnline$ were added. They turn the CCGOLOG interpreter into an on-line mode, or into an off-line mode respectively. This is realized by the fluent $online$. Its successor state axiom is

$$Poss(a,s) \supset (online(do(a,s) \equiv a = setOnline \vee (online(s) \wedge \neg a = clipOnline)).$$

During off-line projections actions like $waitFor$ or continuous fluents are projected according to their least point in time. In the on-line mode, the $waitFor$ becomes an ordinary test action, and continuous fluents are evaluated according to the time that passed by during execution.

One needs a special treatment for updating continuous fluents. A special update action $ccUpdate$ has been introduced for this purpose. The value of a continuous fluent is set to the latest estimate of a low-level process in projections and to the result of the real execution in the on-line case. $ccUpdate$ is realized as an exogenous action which returns the result of the continuous fluent. The action $ccUpdate$ is possible when it occurs after the earliest point in time of the current situation: $Poss(ccUpdate(\vec{x},t),s) \equiv t \geq start(s)$. With $ccUpdate$ the successor state axiom of the fluent $start$ has to be modified:

$$\begin{aligned}
&Poss(a,s) \supset (start(do(a,s)) = t \equiv \\
&\quad \exists\tau.a = waitFor(\tau) \wedge \neg online(s) \wedge ltp(\tau,s,t) \vee \\
&\quad \exists\vec{x}_u.a = ccUpdate(\vec{x}_u,t) \wedge online(s) \vee \\
&\quad (\forall\tau.a \neq waitFor(\tau) \vee online(s)) \wedge (\forall\vec{x}.a \neq ccUpdate(\vec{x}) \vee \neg online(s)) \wedge \\
&\qquad t = start(s)).
\end{aligned}$$

To summarize, CCGOLOG introduces continuous fluents, together with a special fluent update action, a notion of on-line and off-line modes of the interpreter, low-level models of complex tasks, and a system architecture where, from the high-level control point of view, processes can be started in the background.

Reasoning with uncertainty

PGOLOG allows for probabilistic projections over programs. In order to achieve this Grosskreutz (2002) replaces the $Trans$ predicates with functions over probabilities. In particular, he introduces the statement $prob(p,\sigma_1,\sigma_2)$ with the following semantics:

$$\begin{aligned}
&trans(prob(p,\sigma_1,\sigma_2),s,\delta,s') = pr \equiv \\
&\quad \delta = \sigma_1 \wedge s' = do(tossHead,s) \wedge pr = p \vee \\
&\quad \delta = \sigma_2 \wedge s' = do(tossTail,s) \wedge pr = (1-p) \vee \\
&\quad \neg((\delta = \sigma_1 \wedge s' = do(tossHead,s) \vee \\
&\qquad \delta = \sigma_2 \wedge s' = do(tossTail,s)) \wedge pr = 0.
\end{aligned}$$

This means that with probability p the program σ_1 is executed; σ_2 with probability $1-p$. Note that $Trans$ has been replaced by the function $trans : prog \times s \times prog \times s \rightarrow [0,1]$. It holds that

$$trans(\sigma,s,\delta,s') > 0 \text{ iff } Trans(\sigma,s,\delta,s')$$

for all language constructs of CCGOLOG not mentioning $prob$. $tossHead$ and $tossTail$ are dummy actions which are needed to distinguish between the different branches induced by the $prob$ statement. In order to reason with PGOLOG one has to define the transitive closure of $trans$ w.r.t. s:

$$\begin{aligned}
&trans^*(\sigma,s,\delta,s') = p \equiv \forall t[\ldots \supset t(\sigma,s,\delta,s') = p) \vee \\
&\quad p = 0 \wedge \neg\exists p'.\forall t(\ldots \supset t(\sigma,s,\delta,s') = p')
\end{aligned}$$

where the ellipsis stands for the universal closure of the conjunction of the following formulas:

$$\begin{aligned}
&t(\sigma,s,\sigma,s) = 1 \\
&(t(\sigma,s,\sigma^*,s^*) = p_2 \wedge trans(\sigma^*,s^*,\delta,s') = p_1 \wedge p_1 > 0 \wedge p_2 > 0) \\
&\quad \supset t(\sigma,s,\delta,s') = p_1 \cdot p_2
\end{aligned}$$

The formula states that $trans^*$ is equal to the product of the transition probabilities p along its path if there exists a path of at least one transition from configuration $\langle\sigma,s\rangle$ to $\langle\delta,s'\rangle$, or zero otherwise, and that these weights are unique for each path. Grosskreutz (2002) proved that for every model M of the domain axiomatization: $M \models \mathit{trans}^*(\sigma, s, \delta, s') > 0$ iff there exist $\sigma_1, s_1, \dots, \sigma_n, s_n$ such that $\sigma_1 = \sigma$, $s_1 = s$, $\sigma_n = \delta$, $s_n = s'$ and $M \models \mathit{trans}(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ for $i = 1, \dots, n-1$. To be able to probabilistically project PGOLOG programs one has to define the probability that the execution of σ in s results in an execution trace s':

$$\begin{aligned}
&\mathit{doPr}(\sigma, s, s') = p \equiv \\
&\quad \exists \delta. \mathit{trans}(\sigma, s, \delta, s') > 0 \land \mathit{Final}(\delta, s') \land p = \mathit{trans}(\sigma, s, \delta, s') \lor \\
&\quad \neg(\exists \delta. \mathit{trans}(\sigma, s, \delta, s') > 0 \land \mathit{Final}(\delta, s')) \land p = 0.
\end{aligned}$$

We will not define the whole semantics of the statement of PGOLOG here, instead referring to (Grosskreutz 2002). The definitions will be given also in Chapter 4 in the definition of the language READYLOG.

To describe the belief of the agent about the world about a formula φ in a situation s the $\mathit{Bel}(\varphi, s) = p$ is introduced as proposed by (Bacchus et al. 1995). Bel is is an abbreviation for the second-order formula

$$\frac{\sum_{\{s' \mid \varphi[s']\}} p(s', s)}{\sum_{s'} p(s', s)} = p.$$

The projected belief that a sentence φ will hold after a program σ and the low-level model ll_{model} of σ in situation s is defined by

$$\begin{aligned}
&\mathit{PBel}(\varphi, s, \sigma, ll_{model}) = p \overset{def}{=} \\
&\quad \frac{\sum_{\{s', s^* \mid \varphi[s^*]\}} p(s', s) \cdot \mathit{doPr}(\mathit{withPol}(ll_{model}, \sigma), s, s^*)}{\sum_{s'} p(s', s)} = p.
\end{aligned}$$

The result of $\mathit{PBel}(\varphi, s, \sigma, ll_{model})$ represents the weight of all paths that reach a final configuration $\langle \delta^*, s^* \rangle$ where $\varphi[s^*]$ holds normalized by all possible path from s to s'.

Summarizing, PGOLOG introduces a notion of uncertainty with the prob statement. With the assignment of probabilities to programs, probabilistic projections can be defined which yield the probability that a logical formula φ holds in some future situation given by a program σ and a low-level model ll_{model}. In Chapter 5.2.2 we will give an example how this can be used for the decision making of a simulated soccer agent when planning a double pass.

3.3.5 Off-line Decision-theoretic Golog

DTGOLOG was proposed by Boutilier et al. (2000). It extends GOLOG with decision-theoretic planning. Formally, with the domain axiomatization together with an optimization theory one specifies a fully observable finite-horizon MDP $M = \langle S, A, T, R \rangle$ where S is a final set of states, A is a finite set of actions, T is a transition model, and R a real-valued reward function. The set of states of the MDP is implicitly given by the situation terms from the situation calculus, the action set is defined by the domain axiomatization, and the transition model is implicitly defined via the successor state axioms. Additionally, a reward function must be specified.

DTGOLOG then works as follows. It takes a GOLOG program as input and interprets it with an evaluation semantics as given below. Decision-theoretic planning is modeled by nondeterministic choices of actions. At a choice point, DTGOLOG evaluates all possible successor branches according to the optimization theory and inserts the best one into the policy. The policy is a (conditional) GOLOG program where all but the best (nondeterministic) agent choices are optimized away. DTGOLOG implements a forward search value iteration algorithm (cf. Section 3.1.1). The great advantage of DTGOLOG over ordinary value iteration is that it does not rely on an explicit state enumeration. The MDP is induced by the action theory of GOLOG. The reachable states are induced by the successor state axioms. This theoretically allows it to solve MDPs with infinite (continuous) state spaces as only the reachable states will be selected and iterated over.

DTGOLOG introduces a notion for stochastic actions. Reiter's basic action theories do not provide a notion for stochastic actions. It seems like a new sort for stochastic actions has to be introduced. On the other hand, the character of a stochastic actions is such that the agent performs the respective action and nature will choose among the possible outcomes of this action with a certain probability. These outcomes can be regarded as deterministic actions and therefore the basic action theories are extended by only introducing a new predicate $choice(A, n, s)$, where A is a stochastic action, n is one of the outcomes of action A in situation s which nature can choose from. Assuming a finite number m of possible outcomes $N_1, \dots, N_m$ for the stochastic action A under the condition that certain formulas $\varphi(s)$ hold, we can define

$$\begin{aligned} choice(A, a, s) \quad \overset{def}{=} \quad & \varphi_1(s) \supset (a = N_1^1 \vee \cdots N_m^1) \wedge \\ & \vdots \\ & \varphi_k(s) \supset (a = N_1^k \vee \cdots \vee a = N_m^k), \end{aligned}$$

with $\varphi_1(s), \dots, \varphi_k(s)$ a set of mutually disjoint logical conditions which are situation calculus formulas such that $\varphi_1(s) \vee \ldots \vee \varphi_k(s)$ is true for any s. Further, we have to model the probability with which nature chooses a certain outcome. Let $prob(n, a, s)$ denote this probability. For simplicity of notion assume that the outcomes for action A remain the same in each situation, i.e. $choice(A) \overset{def}{=} \{N_1, \dots, N_m\}$. We add the following sentences to the domain axiomatization:

$$prob(N_1, A, s) = p_1, \dots, prob(N_m, A, s) = p_m,$$

where $p_1, \dots, p_m$ are probabilities summing up to 1. If an outcome N_i is not possible in situation s the probability $prob(N_i, A, s)$ must be zero as this outcome cannot occur according to the background theory. Therefore the following must hold:

$$\begin{aligned} & (Poss(N_1, s) \vee \ldots \vee Poss(N_m, s)) \supset \\ & \qquad Poss(N_i, s) \equiv prob(N_i, A, s) > 0,\ i = \{1, \dots, m\}, \\ & (Poss(N_1, s) \vee \cdots \vee Poss(N_m, s)) \supset \sum_{i=1}^{m} prob(N_i, A, s) = 1. \end{aligned}$$

To acquire the assumption of full observability for MDPs one has to extend the action theory by defining formulas $senseCond(n, \varphi)$ which define how the different outcomes of a stochastic action can be discriminated. To sense the state, i.e. evaluate the condition φ in order to determine which outcome has occurred, the sensing action $senseEffect$ is introduced. The axiomatizer of the domain has to take care that φ discriminates the different outcomes.

In the following we give the semantics of DTGOLOG. The semantics is similarly to the evaluation semantics of GOLOG defined as abbreviations of situation calculus formulas.

1. **Zero horizon**
   This is a termination condition for the recursive “calls” of $BestDo$. As DTGOLOG implements a solution algorithm for finite-horizon MDPs, the search for the optimal policy terminates if the remaining horizon reaches zero.

$$\begin{aligned}&BestDo(p,s,h,\pi,v,pr) \overset{def}{=}\\&\quad h=0 \wedge \pi = Nil \wedge v = reward(s) \wedge pr = 1.\end{aligned}$$

2. **The null program**
   If the input program from which the policy is calculated is the nil program, the recursion terminates.

$$\begin{aligned}&BestDo(Nil,s,h,\pi,v,pr) \overset{def}{=}\\&\quad \pi = Nil \wedge v = reward(s) \wedge pr = 1.\end{aligned}$$

3. **Deterministic Action**
   Similar to GOLOG, it is checked whether a primitive action is possible. If the action is not possible, the policy π is terminated with a $Stop$ action, the probability of success pr is set to zero and the value of the policy equals the reward in the current state.[6] If the action is possible, the policy for the remaining program is calculated. The resulting policy is then the primitive action in sequence with the policy for the remaining program, the value is the reward in the actual situation plus the value of the remaining policy.

$$\begin{aligned}&BestDo(a;p,s,h,\pi,v,pr) \overset{def}{=}\\&\quad \neg Poss(a,s) \wedge \pi = Stop \wedge pr = 0 \wedge v = reward(s)\\&\quad \vee Poss(a,s) \wedge \exists(\pi',v',pr').BestDo(p,do(a,s),h-1,\pi',v',pr')\\&\qquad \wedge \pi = a;\pi' \wedge v = reward(s)+v' \wedge pr = pr'\end{aligned}$$

4. **Stochastic action**
   In the case of a stochastic action, the predicate $BestDoAux$ with the set of all outcomes for this stochastic action is expanded. We use as Soutchanski (2003) $choice'(a) \overset{def}{=} \{n_1,\ldots,n_k\}$ as an abbreviation for the outcomes of the stochastic action a.

$$\begin{aligned}&BestDo(a;p,s,h,\pi,v,pr) \overset{def}{=}\\&\quad \exists \pi',v'.BestDoAux(choice'(a),a,p,s,h,\pi',v',pr) \wedge\\&\quad \pi' = a;senseEffect(a);\pi' \wedge v = reward(s)+v'\end{aligned}$$

[6]Note that we use the terms state and situation synonymously. We specify when a distinction has to be made.

The resulting policy is $a; \mathit{senseEffect}(a); \pi'$. The pseudo action $\mathit{senseEffect}$ is introduced to fulfill the requirement of full observability. The remainder policy π' branches over the possible outcomes and the agent must be enabled to sense the state it is in after having executed this action. The remainder policy is evaluated using the predicate $BestDoAux$. The predicate $BestDoAux$ for the (base) case that there is one outcome is defined as

$$\begin{aligned}
& BestDoAux(\{n_k\}, a, \delta, s, h, \pi, v, pr) \overset{def}{=} \\
& \quad \neg Poss(n_k, s) \wedge \pi = \mathit{Stop} \wedge v = 0 \wedge pr = 0 \vee \\
& \quad Poss(n_k, s) \wedge senseCond(n_k, \varphi_k) \wedge \\
& \qquad \exists \pi', v', pr'.\mathit{BestDo}(\delta, do(n_k, s), h, \pi', v', pr') \wedge \\
& \qquad\quad \pi = \varphi_k?; \pi' \wedge v = v' \cdot \mathit{prob}(n_k, a, s) \wedge pr = pr' \cdot \mathit{prob}(n_k, a, s)
\end{aligned}$$

If the outcome action is not possible, the Stop action is inserted into the policy and no further calculations are conducted. Otherwise, if the current outcome action is possible, the remainder policy π' for the remaining program is calculated. The policy π consists of a test action on the condition φ_k from the $senseCond$ predicate with the remainder policy π' attached. The case for more than one remaining outcome action is defined as

$$\begin{aligned}
& BestDoAux(\{n_1, \ldots, n_k\}, a, p, s, h, \pi, v, pr) \overset{def}{=} \\
& \quad \neg Poss(n_1, s) \wedge BestDoAux(\{n_2, \ldots, n_k\}, p, s, h, \pi, v, pr) \vee \\
& \quad Poss(n_1, s) \wedge (\exists \pi', v', pr').BestDoAux(\{n_2, \ldots, n_k\}, p, s, h, \pi', v', pr') \wedge \\
& \quad \exists \pi_1, v_1, pr_1.\mathit{BestDo}(p, do(n_1, s), h-1, \pi_1, v_1, pr_1) \wedge senseCond(n_1, \varphi_1) \\
& \quad \pi = \textbf{if } \varphi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \textbf{ endif} \wedge \\
& \quad v = v' + v_1 \cdot \mathit{prob}(n_1, a, s) \wedge pr = pr' + p_1 \cdot \mathit{prob}(n_1, a, s)
\end{aligned}$$

The difference to the previous $BestDoAux$ predicate is that the other outcomes are recursively interpreted, and that the resulting policy now consists of a conditional instead of a test action as in the previous case. The value for the outcome is clearly the value of the remaining policy which has n_1 as prefix weighted by the probability of occurrence of n_1 plus the value gathered by the other possible outcomes. Similarly the probability of success pr is calculated.

5. **Test Action**
A test action is similar to GOLOG despite that π, v, and pr have to be instantiated appropriately. Similar to a deterministic action, Stop is inserted in the case that the test condition does not hold and the calculation of the policy is terminated.

$$\begin{aligned}
& \mathit{BestDo}(a; p, s, h, \pi, v, pr) \overset{def}{=} \\
& \quad \phi[s] \wedge \mathit{BestDo}(p, s, h, \pi, v, pr) \vee \\
& \quad \neg\phi[s] \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = \mathit{reward}(s)
\end{aligned}$$

6. **Nondeterministic Choice of Actions**
   Nondeterministic choices of actions allow for DT planning. For both choices the continuation policies π_1 and π_2 are calculated. A multi-criteria analysis over the values and the probability of success is then done and the optimal policy is returned.

$$
\begin{aligned}
&BestDo((p_1|p_2);p,s,h,\pi,v,pr) \overset{def}{=} \\
&\quad \exists\pi_1,v_1,pr_1.BestDo(p_1;p,s,h,\pi_1,v_1,pr_1) \wedge \\
&\quad \exists\pi_2,v_2,pr_2.BestDo(p_2;p,s,h,\pi_2,v_2,pr_2) \wedge \\
&\quad ((v_1,p_1) \geq (v_2,p_2) \wedge \pi = \pi_1 \wedge pr = pr_1 \wedge v = v_1) \vee \\
&\quad (v_1,p_1) < (v_2,p_2) \wedge \pi = \pi_2 \wedge pr = pr_2 \wedge v = v_2)
\end{aligned}
$$

7. **Conditional**
   A conditional is like in GOLOG an abbreviation for a test action and the respective branches of the conditional.

$$
\begin{aligned}
&BestDo((\mathbf{if}\ \varphi\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{endif};p),s,h,\pi,v,pr) \overset{def}{=} \\
&\quad BestDo(((\varphi?;p_1)|(\neg\phi?;p_2);p),s,h,\pi,v,pr)
\end{aligned}
$$

8. **Nondeterministic Finite Choice of Arguments**
   DTGOLOG allows for optimal choices over action arguments. In the program p, all free variables x are substituted by τ. The domain of $\tau = \{v_1,\ldots,v_n\}$ must be finite. An optimization is initiated for each substitution, the best argument is chosen for the policy. We also refer to this statement as $pickBest$ as a optimized version of the GOLOG "$pick$" construct.[7]

$$
\begin{aligned}
&BestDo((\varpi(x:\tau)p);p',s,h,\pi,v,pr) \overset{def}{=} \\
&\quad BestDo(p|^x_{c_1}\cdots p|^x_{c_n});p',s,h,\pi,v,pr)
\end{aligned}
$$

9. **Sequential Composition**
   Sequential composition is the same as in GOLOG.

$$
\begin{aligned}
&BestDo((p_1;p_2);p_3,s,h,\pi,v,pr) \overset{def}{=} \\
&\quad BestDo(p_1;(p_2;p_3),s,h,\pi,v,pr).
\end{aligned}
$$

Loops and procedures are not given a formal semantics in Soutchanski (2003). He remarks that these constructs require a second-order definition and refers to the implementation of his DTGOLOG interpreter (Soutchanski 2003, Appendix C.1).

[7] In GOLOG the "pick" statement is abbreviated with π. This should not be confused with the variable π for policies in the decision-theoretic context.

To illustrate how DTGOLOG calculates an optimal policy we give an example in Chapter 4.2. Boutilier et al. (2000) give an example from a service robotics domain. The robot's objective is to deliver mail in an office environment. The domain axiomatization comprises fluents like $hasMail(person, s)$, $mailPresent(person, m, s)$, $robotLoc(loc, s)$.

The DTGOLOG program which calculates the optimal policy for delivering the mails is

```
proc Mail
  while $\exists p.\neg attempted(p) \wedge \exists n.mailPerson(p, n))$ do
    $\varpi(p, people,$
      $(\neg attempted(p) \wedge \exists n.mailPresent(p, n); deliverTo(p)))$
    endwhile
  endproc
```

$deliverTo$ is a complex procedure including picking up letters from the post box, moving to a person's office, handing over the letter, and returning to the post box. Each person in the domain has a different priority for mail being delivered to them. The reward the robot receives for his actions is discounted depending on the priority of the person and on the time it approximately takes to deliver the mail. The condition $\neg attempted(p)$ serves as a guard condition here.

The state space of this domain having P people, L locations and N letters is about $2^N \cdot (6 \cdot N + 6)^P \cdot L^3$. The authors state that a problem instance with 5 people and 7 locations would demand 2.7 billion states for ordinary value iteration. In DTGOLOG this problem size can still be handled.

In his Ph.D. thesis Soutchanski (2003) compares DTGOLOG with the state-of-the-art MDP solver SPUDD (Hoey et al. 1999). He shows that DTGOLOG solves problem instances faster than SPUDD. The reason lies in the forward search value iteration algorithm which is used by DTGOLOG (by means of $BestDo$). Only the relevant successor states are expanded instead of SPUDD which has to iterate over all states of the MDP. The optimal policies on a given problem instance are the same w.r.t. the states expanded by DTGOLOG (clearly, DTGOLOG cannot provide an optimal action for states which where not visited; SPUDD on the other hand provides optimal actions for all states) and the resulting values are qualitatively the same with both approaches. This gives further evidence for the usefulness of DTGOLOG as solver for MDPs.

3.4 Summary

In this section we illuminated the mathematical background for the methods we need throughout this thesis. For one these are solution methods for Markov Decision Processes. We introduced the mathematical model for MDPs and showed several solution methods. In our definition we left out cost models. These can be easily integrated. A whole body of different models exist, derived from the basic model we presented in Section 3.1 like POMDPs or semi-MDPs. There are also other optimization criteria than the expected cumulated reward which is the optimization criterion we showed here, like the average reward. We presented some core solution techniques for finding optimal policies. These methods are called decision-theoretic planning methods. Finally, we showed the relationship between reinforcement learning as a technique to find optimal policies if the transition model is not known. Decision-theoretic planning will be a topic in Chapter 4. A thorough mathematical treatment of MDPs and the different model variants can be found in (Puterman 1994).

Bayes filter (Section 3.2) are probabilistic methods for solving hidden Markov models. In the context of this thesis we regard Bayes filter for the localization problem of a mobile robot. The state of the system cannot be observed directly. It has to be estimated by observations, i.e. sensor values. We showed the mathematical background of the Kalman filter and a Monte Carlo localization approach, which uses sampling techniques to estimate the pose of the robot. While Kalman filter are Bayes filter with a uni-model Gaussian representation of the belief distribution, Monte Carlo techniques have a multi-modal distribution. Hence, Kalman filter basically cannot represent multiple hypothesis for the pose of the robot. We will use these models throughout Chapter 6.

Finally in Section 3.3, we showed the important mathematical background the situation calculus and GOLOG. Our approach with READYLOG is founded on these action formalisms. We introduced the GOLOG derivatives which we integrated into READYLOG in detail. We want to refer to Reiter's textbook on the situation calculus and GOLOG (Reiter 2001) for further reading.

Chapter 4

A Golog Dialect for Real-time Dynamic Domains

The aim of designing the language READYLOG is to create a GOLOG dialect which supports the programming of the high-level control of agents or robots in dynamic real-time domains. The primary application has been robotic soccer. The robotic soccer domain has some specific characteristics which made the development of READYLOG necessary and influenced several design decisions: the robotic soccer domain is an unpredictable adversarial dynamic real-time domain. This means that decisions have to be taken quickly and making plans for future courses of actions have a mid-term horizon. Planning ahead for the next minute does not make sense as the world changes unpredictably due to the uncertainty of the outcomes of own actions and the behaviors of the opposing players. The unpredictability of the actions of the agent calls for some notion of uncertainty. Actions succeed with a certain probability p or fail with a probability of $1 - p$. Further, the soccer domain is a continuous domain, whereas the world in the situation calculus evolves from situation to situation. The agent programming language has to support a continuously changing world. The complexity of the domain also has an effect on the aspect of how to program the agents in practice. The idea of GOLOG is to combine agent programming with planning. Usually this means that a certain goal should be reached and it is proved that with the actions programmed the goal can be reached. As it is generally not obvious to the programmer which series of actions will end in scoring a goal, it seems helpful to use an optimization theory to evaluate different action choices and execute the most promising one w.r.t. the success probability and the optimization theory. Several other aspects influenced the language READYLOG. The knowledge the robot or agent has about its environment is incomplete. This means that the robot has to use its sensors permanently to gather knowledge about the environment. When the robot has to query its sensors frequently it becomes an issue as to how the new knowledge can be integrated fast enough.

Several extensions of the original GOLOG dialect exist which cover specific areas. De Giacomo and Levesque (1999), De Giacomo et al. (2002) proposed an incremental on-line GOLOG interpreter (INDIGOLOG), where actions are directly executed in the real world. Grosskreutz and Lakemeyer (2000b) and Grosskreutz (2000) proposed pGOLOG which extends GOLOG with probabilistic programs. With a certain specified probability p a program σ succeeds, or fails with probability $1 - p$. A semantics to deal with continuous change was proposed with CCGOLOG in (Grosskreutz and Lakemeyer 2000a), sensing was presented in (Grosskreutz and Lakemeyer

	On-line	Sensing	Exog. Actions	Conc. Exec.	Projection	Prob. Actions	Cont. Change	Decision Theory
Golog	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
ConGolog	$-$	$-$	$-$	$+$	$-$	$-$	$-$	$-$
IndiGolog	$+$	$+$	$+$	$+$	$+$	$-$	$-$	$-$
ccGolog	$+$	$+$	$+$	$+$	$+$	$-$	$+$	$-$
pGolog	$+$	$+$	$+$	$-$	$+$	$+$	$-$	$-$
DTGolog	$-$	$-$	$-$	$-$	$+$	$+$	$-$	$+$
Readylog	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$

Table 4.1: Features of Golog Languages

2001). Boutilier et al. (2000) proposed DTGOLOG, a decision-theoretic GOLOG dialect which uses an MDP based optimization theory to find the optimal course of actions. Table 4.1 gives an overview of the extensions of GOLOG. The achievement of READYLOG is to integrate those aspects into one language and framework. In particular, these are the continuous fluents and the model of concurrency from CCGOLOG, the projection mechanism from PGOLOG, and the integration of decision-theory from DTGOLOG. Several modifications and extensions have been made:

- a novel on-line version of the decision-theoretic planning method proposed by Boutilier et al. (2000), which allows for execution monitoring of policies;
- the introduction of macro actions, so-called options, for decision-theoretic planning after an idea of Precup et al. (1998);
- an enhanced version of passive sensing which allows for update a whole world model in one sweep;
- several speed-ups for policy generation as making use of caching previously computed results in the forward decision-theoretic search for an optimal policy,
- a useful any-time approach for decision-theoretic planning to overcome fixed horizons when searching for a policy and by this to better exploit the computational resources of the agent or robot, and
- a progression method after Lin and Reiter (1997).

The core language is presented in Section 4.1. Here, we give a brief overview of the semantics of the language. We will not go into the details, as most of the material is presented in the literature or in Chapter 3. We give pointers to the original resources. Section 4.2 presents our approach to on-line decision-theoretic planning in detail. We relate our approach to another approach to on-line DT planning by Soutchanski (2001) and show why his approach is not applicable for the domains we are aiming at. In Section 4.3 we show our approach to use macro actions for speeding up the computation of decision-theoretic planning. First, we illustrate the theoretical background of macro actions in the context of Markov Decision Processes, before we present our solution algorithm. Further, we show how some simple, but efficient techniques further speed up the process of policy generation. Section 4.4 addresses implementation details of the interpreter and shows the progression method. We conclude with a discussion in Section 4.5.

4.1 Readylog Semantics

4.1.1 Overview

READYLOG borrows the approach to sensing and on-line execution from INDIGOLOG. This means that READYLOG uses the approach of (sensing) histories for active sensing actions. Sensed values are entered into the action history as described in Chapter 3.3.3. Besides this active sensing approach, READYLOG makes use of the concept of passive sensing. This was proposed by (Grosskreutz and Lakemeyer 2001) and was discussed in Chapter 3.3.4 as a feature of CCGOLOG. To recap, a world model update for continuous fluents is conducted not by an active sensing action, but as an exogenous event that is raised when the special $ccUpdate$ action is performed.

This concept was extended in READYLOG. When passive sensing updates come in frequently it is more efficient to update all fluents at once. We discuss our extension in Section 4.2.2. This brings us to the next feature of READYLOG: continuous change. For continuous domains like robotic soccer we are aiming at the integration of continuous fluents is indispensable. Together with these special fluents we also use the idea of the control architecture employing low-level models and action registers as proposed in (Grosskreutz 2002) and discussed in Chapter 3.3.4. Further, we integrated the possibility to use probabilistic projections from PGOLOG. This also requires us to make use of a weighted transition semantics which we will present in Section 4.1.3.

Finally, our READYLOG dialect integrates decision-theoretic planning. The foundations of DT-GOLOG and its semantics were given in Chapter 3.3.5. This also includes the notion of stochastic actions which is different from the representation of PGOLOG. To ease implementation of large programs and complex effects we extend the notion of stochastic actions from the notion presented in Chapter 3.3.5 in such a way that it is possible to encode outcomes of a stochastic action by sequences and conditionals over primitive actions. We introduce our extension in Section 4.2.4.

To make decision-theoretic planning available in the READYLOG framework we propose an on-line execution system for policies which is also able to monitor when policies become invalid due to unforeseen changes in the environment. Another feature of READYLOG are the so-called options. These are macro-actions in the decision-theoretic context. We show how these are integrated into the READYLOG framework and show their usefulness for discrete domains.

4.1.2 Reifying Programs as Terms

The language GOLOG is defined as abbreviations of situation calculus formulas. Each program statement is expanded into a logical formula of the situation calculus. However, with the introduction of the transition semantics the program statements can no longer be defined in terms of abbreviations of the logic. They have to be defined as terms of logic itself. This requires to reify programs in the language as terms.

For reifying programs as terms in the language of the situation calculus one has to formally define formulas as terms. As we make use of continuous change and the notion of probabilistic programs, we further have to introduce the sorts real, time, prob, formula, continuous formula, tform.

We will not give the definitions of all sorts and how terms of the respective sorts are interpreted in detail. Instead we sketch the ideas of these definitions and refer to (Grosskreutz 2002) for a complete definition of the reification of CCGOLOG and PGOLOG.

First of all, one has to define an index structure isomorphic to the natural numbers for referring to arguments of terms. Then, one has to formally introduce the different sorts like situation, object, or action into the language and define variables and functions on these sorts. An example is the function $\mathsf{nameOf}_{Sort} : Sort \rightarrow PseudoSort$, which is a mapping from situation calculus sorts to the reified version of the sorts, which are called $PseudoSort$. The function $\mathsf{var}_{Sort} : Idx \rightarrow PseudoSort$ defines a mapping from the index structure to a reified sort. It is used to reference variables in arguments. For each function $f : Sort_1 \times \cdots \times Sort_n \rightarrow Sort_{n+1}$ of the situation calculus a mapping $\mathsf{f} : PseudoSort_1 \times \cdots \times PseudoSort_n \rightarrow PseudoSort_{n+1}$ has to be defined to reference the reified versions of functions. For predicates of a particular situation calculus sort a reified version is defined together with a set of domain closure and unique names axioms. To relate functions of pseudo-sorts which do not mention pseudo variables to real sorts a predicate Closed is defined. The closed terms are $\mathsf{Closed}(now)$, $\mathsf{Closed}(\mathsf{nameOf}(x))$, $\neg\mathsf{Closed}(\mathsf{z}_i)$, and for each f $\mathsf{Closed}(\mathsf{f}(x_1, \ldots, x_n) \equiv \mathsf{Closed}(x_1) \wedge \cdots \wedge \mathsf{Closed}(x_n)$. A function $\mathsf{decode} : PseudoSort \times Sit \rightarrow Sort$ accomplishes this. For example, $\mathsf{decode}(now, s) = s$, $\mathsf{decode}(\mathsf{nameOf}(x)) = x$, or $\mathsf{decode}(\mathsf{f}(x_1, \ldots, x_n), s) = f(\mathsf{decode}(x_1, s), \ldots, \mathsf{decode}(x_n, s))$.

Next, one has to define a functional mapping for logical formulas. This mapping is inductively defined over the structure of the logical formula, defining a mapping of the predicate itself to a function, conjunction, negation and existential quantification are also mapped. Variable substitution is defined for the arguments of a predicate. Similar to the decode function a function $\mathsf{Holds} : PseudoSit \times Sit$ is defined to encode the truth value of a predicate. For example, $\mathsf{Holds}(\mathsf{p}(x_1, \ldots, x_n), s) \equiv p(\mathsf{decode}(x_1, s), \ldots, \mathsf{decode}(x_n, s))$, or $\mathsf{Holds}(\exists \mathsf{z}.\rho, s) \equiv \exists y.\mathsf{Holds}(\rho^{\mathsf{z}}_{\mathsf{nameOf}(y)}, s)$. A notation we use in the semantic definition of READYLOG is $\varphi[s]$ and denotes $\mathsf{Holds}(\varphi, s)$. Similarly, function mappings for continuous fluents, and probabilistic projections are defined to relate situation calculus sorts to their reified pseudo-sorts versions. Finally, functional mappings for programs are defined by introducing a sort $Prog$ and a mapping from program statements to the sort program. For example, a primitive action is defined by $\mathsf{act} : PseudoAct \rightarrow Prog$, or a sequence of program statements by $\mathsf{seq} : PseudoFormula \rightarrow Prog$. The notions introduced in the semantic definitions are therefore notions which denote the functions of sort $Prog$.

It was shown ((De Giacomo et al. 2000) for CONGOLOG; (Grosskreutz 2002) for CCGOLOG and PGOLOG) that the above described encoding preserves consistency. We refer to (Grosskreutz 2002) for complete definition for reifying programs as terms of the logical language. Note that in the following we use the logical notation of $Sort$. These must be seen as a notation for their reified version as sketched above.

4.1.3 Trans and Final

In the following we define the semantics for the core language. We make use of the weighted transition semantics introduced by (Grosskreutz 2002). We omit the somewhat lengthy second order definition of the transition semantics with procedures and refer to (Grosskreutz 2002) for details. Other resources for the definition of procedures in the transition semantics context are (De Giacomo et al. 1997; De Giacomo et al. 2000)

Instead of using macro expansion as GOLOG does, we use a transition semantics introduced by (De Giacomo et al. 2000). It defines program execution by a one step transition between program configurations. A configuration is a tuple $\langle \gamma, s\rangle$, where γ is a program and s a situation. The special predicate $trans(\gamma, s, \delta, s')$ transforms program γ in the situation s into the program δ resulting in the situation s'. To denote final configurations, i.e. such configuration where the computation of a statement is finished, a predicate $Final(\gamma, s)$ exists. Originally, (De Giacomo et al. 2000) defined $trans$ as a predicate. Grosskreutz (Grosskreutz 2002) extended program transitions to return the probability of a step in program execution to account for probabilistic program execution. He defined a weighted transition semantics for pGOLOG constructs. Thus, $trans : program \times situation \times program \times situation \rightarrow [0, 1]$ is a function which maps pairs of configurations to probabilities. READYLOG integrates pGOLOG and the possibility to weight programs with probabilities. Therefore, the semantics of the READYLOG constructs make also use of the weighted transition semantics proposed in (Grosskreutz 2002). For readability, we write

$$f(\vec{y}) = \textbf{if } (\exists \vec{y}).\varphi(\vec{x}, \vec{y}) \textbf{ then } g(\vec{x}, \vec{y}) \textbf{ else } h(\vec{x})$$

as an abbreviation for

$$f(\vec{x}) = p \equiv (\exists \vec{y}).\varphi(\vec{x}, \vec{y}) \wedge p = g(\vec{x}, \vec{y}) \vee \neg(\exists \vec{y}).\varphi(\vec{x}, \vec{y}) \wedge p = h(\vec{x})$$

where $\varphi(\vec{x}, \vec{y})$ is a first-order formula with free variables among $\vec{x} \cup \vec{y}$, and similarly $g(\vec{x}, \vec{y})$ and $h(\vec{x})$ are functions whose arguments range over $\vec{x} \cup \vec{y}$ and $\vec{x}$, resp. Similarly,

$$f(\vec{x}) = \textbf{if } (\exists \vec{y_1}).\varphi_1(\vec{x}, \vec{y_1}) \textbf{ then } g_1(\vec{x}, \vec{y_1}) \textbf{ elseif} (\exists \vec{y_2}).\varphi_2(\vec{x}, \vec{y_2}) \textbf{ then } g_2(\vec{x}, \vec{y_2}) \textbf{ else } h(\vec{x})$$

is an abbreviation for

$$\begin{aligned} f(\vec{x}) = p \equiv{}& (\exists \vec{y_1}).\varphi_1(\vec{x}, \vec{y_1}) \wedge p = g_1(\vec{x}, \vec{y_1}) \vee (\exists \vec{y_2}).\varphi_2(\vec{x}\vec{y_2}) \wedge p = g_2(\vec{x}\vec{y_2}) \vee \\ & \neg((\exists \vec{y_1}.\varphi_1(\vec{x}, \vec{y_1}) \vee (\exists \vec{y_2}).\varphi_2(\vec{x}, \vec{y_2})) \wedge p = h(\vec{x}) \end{aligned}$$

Figure 4.1 gives an overview of the READYLOG constructs which we will present in the following.

Empty Program

$$trans(nil, s, \delta, s') = 0$$
$$Final(nil, s) \equiv true$$

The empty program contributes a probability of 0 to the current total probability of the program branch. Intuitively this means that no further transition is possible. At the same time the empty program reaches a final configuration.

Primitive Action

$$trans(\alpha, s, \delta, s') = \textbf{if } Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s) \textbf{ then } 1 \textbf{ else } 0$$
$$Final(\alpha, s) \equiv false$$

If the program consists of a primitive action α, it is checked whether the action is possible w.r.t. the background action theory denoted by $Poss(\alpha[s], s)$. If this holds, the program is transformed to the successor configuration $\langle nil, do(\alpha[s], s)\rangle$. The probability for this transition is obviously 1. If the primitive action α is not possible this transition gets the value 0 which is equivalent to false.

nil	empty program
α	primitive action
$\varphi?$	wait/test action
$waitFor(\tau)$	event-interrupt
$[\sigma_1; \sigma_2]$	sequence
$\mathbf{if}\ \varphi\ \mathbf{then}\ \sigma_1\ \mathbf{else}\ \sigma_2\ \mathbf{endif}$	conditional
$\mathbf{while}\ \varphi\ \mathbf{do}\ \sigma\ \mathbf{endwhile}$	loop
$\mathbf{withCtrl}\ \varphi\ \mathbf{do}\ \sigma\ \mathbf{endwithCtrl}$	guarded execution
$\sigma_1 \,\|\|\, \sigma_2$	prioritized execution
$\mathbf{forever\ do}\ \sigma\ \mathbf{endforever}$	infinite loop
$\mathbf{whenever}(\tau, \sigma)$	interrupt triggered by continuous function
$\mathbf{withPol}(\sigma_1, \sigma_2)$	prioritized execution until σ_2 ends
$\mathbf{prob}(p, \sigma_1, \sigma_2)$	probabilistic execution of either σ_1 or σ_2
$\mathbf{interrupt}$	interrupts
$pproj(c, \sigma)$	probabilistic (off-line) projection
$\{\mathbf{proc}\ P_1(\vec{\vartheta}_1)\sigma_1\ \mathbf{endproc}; \cdots; \mathbf{proc}\ P_n(\vec{\vartheta}_n)\sigma_n\ \mathbf{endproc}\}; \sigma_0$	procedures
$solve(\sigma, h)$	initiate decision-theoretic optimization over σ
$\sigma_1 \,\|\, \sigma_2$	nondeterministic (dt) choice of programs
$pickBest(\vec{x}, \sigma, h)$	nondeterministic (dt) choice of arguments

Figure 4.1: Overview of Readylog constructs

Test Action

$$\begin{aligned}&trans(\varphi?, s, \delta, s') = \mathbf{if}\ \varphi[s] \wedge \delta = nil \wedge s' = s\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\ &Final(\varphi?, s) \equiv false\end{aligned}$$

Similar as for primitive actions, the remaining program is $\delta = nil$. Note that unlike primitive actions the successor situation remains the same as a test does not change the environment. The transition returns the value 1 or 0, resp., depending on whether the test condition $\varphi[s]$ holds. A test action does not reach a final configuration.

Event-Interrupt

To achieve event-driven behavior a notion of time must be integrated into the language. This was introduced by Grosskreutz (2002, Chpt. 4) and discussed in Chapter 3.3.4. To represent the value of continuous fluents so-called $t\text{-}function$ have been introduced which represents functions of time. These are needed to project the value of continuous fluents w.r.t. the time that has passed. The argument τ of a $waitFor(\tau)$ statement is restricted to $t\text{-}forms$. The idea behind $waitFor(\tau)$ is that actions should happen as soon as possible. Thus, the notion of a least time point $ltp(\tau, s, t)$ is needed. It is an abbreviation for the formula

$$\tau[s,t] \wedge t \geq start(s) \wedge \forall t'.start(s) \leq t' < t \supset \neg[s, t']$$

which means that t in $ltp(\tau, s, t)$ is the least point in time after the start of s where τ becomes true. To evaluate a *t-form* in a situation s at a time t we write $\tau[s,t]$ which results in a formula which is like τ except that every continuous fluent F is replaced by $val(F(s),t)$, which stands for the t-$function$ assigned to fluent F. $trans$ and $Final$ is then defined as

$$\begin{aligned}
&trans(waitFor(\tau), s, \delta, s') = p \equiv \\
&\quad p = 1 \wedge (online[s] \wedge \tau[s,t] \wedge s' = s \wedge \delta = nil \vee \\
&\quad \neg online[s] \wedge ltp(\tau, s, t) \wedge s' = do(setTime(t), s) \wedge \delta = nil) \\
&Final(waitFor(\tau), s) \equiv false
\end{aligned}$$

This definition means that in the on-line case[1] the t-$form$ is evaluated. No further transition is taken. If $\tau[s,t]$ does not hold, the interpreter will wait forever. In the off-line case (projection) the least time point is calculated and the time is advanced accordingly.

Sequence

$$\begin{aligned}
&trans(\sigma_1;\sigma_2, s, \delta, s') = \\
&\quad \textbf{if } (\exists\gamma).\delta = \gamma;\sigma_1 \wedge trans(\sigma_1, s, \gamma, s') > 0 \textbf{ then } trans(\sigma_1, s, \gamma, s') \\
&\quad \textbf{elseif } Final(\sigma_1, s) \wedge trans(\sigma_2, s, \delta, s') > 0 \textbf{ then } trans(\sigma_2, s, \delta, s) \textbf{ else } 0 \\
&Final(\sigma_1;\sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)
\end{aligned}$$

Sequences of actions are transformed in the following way: either there is a successor configuration for σ_1 (with positive weight) or σ_1 has already become final. In the latter case the program σ_2 must be further transformed. A sequence of programs is regarded as final if both programs reach a final configuration.

Conditional

$$\begin{aligned}
&trans(\textbf{if } \varphi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2 \textbf{ endif}, s, \delta, s') = \\
&\quad \textbf{if } \varphi[s] \textbf{ then } trans(\sigma_1, s, \delta, s') \textbf{ else } trans(\sigma_2, s, \delta, s') \\
&Final(\textbf{if } \varphi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2 \textbf{ endif}, s) \equiv \\
&\quad \varphi[s] \wedge Final(\sigma_1, s) \vee \neg\varphi[s] \wedge Final(\sigma_2, s)
\end{aligned}$$

Depending on the condition φ the remaining program σ_1 or σ_2, resp., is executed. The function value (probability) of this transition depends on the return value of either σ_1 or σ_2. A conditional is final when the respective branch to be executed reaches a final configuration.

Loop

$$\begin{aligned}
&trans(\textbf{while } \varphi \textbf{ do } \sigma \textbf{ endwhile}, s, \delta, s') = \\
&\quad \textbf{if } (\exists\gamma).\varphi[s] \wedge \delta = \gamma; \textbf{while } \varphi \textbf{ do } \sigma \textbf{ endwhile then } trans(\sigma, s, \gamma, s') \textbf{ else } 0 \\
&Final(\textbf{while } \varphi \textbf{ do } \sigma \textbf{ endwhile}, s) \equiv \neg\varphi[s] \vee Final(\sigma, s)
\end{aligned}$$

[1] $online$ is a built-in fluent keeping track of the operation mode of the interpreter.

A loop is transformed by first testing the loop condition φ. If there is a configuration γ which results from the successor configuration of the loop body σ, the remaining program to be further transformed is the successor program γ with the loop attached. This ensures that the loop is being executed as long as the condition φ holds. A loop reaches a final configuration if the condition does not hold or the loop body reaches a final configuration.

Guarded Execution

$$\begin{aligned}
&\mathit{trans}(\mathbf{withCtrl}\,\varphi\,\mathbf{do}\,\sigma\,\mathbf{endwithCtrl}, s, \delta, s') = \\
&\quad \mathbf{if}\ (\exists\gamma).\varphi[s] \land \delta = \mathbf{withCtrl}\,\varphi\,\mathbf{do}\,\gamma\,\mathbf{endwithCtrl} \\
&\quad \mathbf{then}\ \mathit{trans}(\sigma, s, \gamma, s')\ \mathbf{else}\ 0 \\
&\mathit{Final}(\mathbf{withCtrl}\,\varphi\,\mathbf{do}\,\sigma\,\mathbf{endwithCtrl}, s) \equiv \varphi[s] \land \mathit{Final}(\sigma, s)
\end{aligned}$$

The idea behind guarded execution is to execute program σ, the body of the guard, as long as the guard condition φ holds. It is final when the body reaches a final condition and at the same time the condition holds.

Prioritized Execution

$$\begin{aligned}
&\mathit{trans}(\sigma_1 \,||\, \sigma_2, s, \delta, s') = \\
&\quad \mathbf{if}\ \neg\mathit{Final}(\sigma_1, s) \land \neg\mathit{Final}(\sigma_2, s) \land \exists\gamma_1.\mathit{trans}(\sigma_1, s, \gamma_1, s') > 0 \land \delta = \gamma_1 \,||\, \sigma_2 \land \\
&\qquad (\forall\gamma_2, s_2).\mathit{trans}(\sigma_2, s, \gamma_2, s_2) > 0 \supset \mathit{start}(s') \leq \mathit{start}(s_2) \\
&\quad \mathbf{then}\ \mathit{trans}(\sigma_1, s, \gamma_1, s') \\
&\quad \mathbf{elseif}\ \neg\mathit{Final}(\sigma_1, s) \land \neg\mathit{Final}(\sigma_2, s) \land (\exists\gamma_2).\mathit{trans}(\sigma_2, s, \gamma_2, s') > 0 \land \delta = \sigma_1 \,||\, \gamma_2) \land \\
&\qquad (\forall\gamma_1, s_1).\mathit{trans}(\sigma_1, s, \gamma_1, s_1) > 0 \supset \mathit{start}(s') < \mathit{start}(s) \\
&\quad \mathbf{then}\ \mathit{trans}(\sigma_2, s, \gamma_2, s')\ \mathbf{else}\ 0 \\
&\mathit{Final}(\sigma_1 \,||\, \sigma_2, s) \equiv \mathit{Final}(\sigma_1, s) \lor \mathit{Final}(\sigma_2, s)
\end{aligned}$$

The idea of prioritized execution is that program σ_1 is executed with priority over program σ_2 if σ_1 can be executed before σ_2 (w.r.t. execution time). In the above definition the special fluent *start* encodes when a program started to be executed. When both σ_1 and σ_2 are not in a final configuration and σ_1 can be transformed into program γ_1 and the execution of σ_1 can be started before any of the possible successor configurations of σ_2, the remaining program is $\gamma_1 \,||\, \sigma_2$, i.e. the first step of σ_1 is prioritized over σ_2. Otherwise, σ_2 will be started first. A final configuration is reached when either σ_1 or σ_2 becomes final.

Infinite Loop

$$\mathbf{forever\ do}\ \sigma\ \mathbf{endforever} \overset{def}{=} \mathbf{while}\ \mathit{true}\ \mathbf{do}\ \sigma\ \mathbf{endwhile}$$

forever is a useful abbreviation for an infinite loop.

Interrupt Triggered by Continuous Function

$$\mathbf{whenever}(\tau, \sigma) \stackrel{def}{=} \mathbf{forever}([waitFor(\tau, \sigma)])$$

The $\mathbf{whenever}$ statement is an abbreviation for a guarded infinite loop.

Prioritized Execution until σ_2 ends

$$\mathbf{withPol}(\sigma_1, \sigma_2) \stackrel{def}{=} (\sigma_1; \mathit{false}?) \,||\, \sigma_2$$

σ_1 takes priority over σ_2. The test $\mathit{false}?$ averts that σ_1 becomes final. This means that the whole construct can only become final if σ_1 was executed up to the end and σ_2 became final.

Probabilistic Execution

$$\begin{aligned}
&\mathit{trans}(\mathbf{prob}(p, \sigma_1, \sigma_2), s, \delta, s') = \\
&\quad \mathbf{if}\ \delta = \sigma_1 \wedge s' = \mathit{do}(\mathit{tossHead}, s)\ \mathbf{then}\ p \\
&\quad \mathbf{elseif}\ \delta = \delta_2 \wedge s' = \mathit{do}(\mathit{tossTail}, s)\ \mathbf{then}\ 1 - p\ \mathbf{else}\ 0 \\
&\mathit{Final}(\mathbf{prob}(p, \sigma_1, \sigma_2), s) \equiv \mathit{false}
\end{aligned}$$

The $\mathbf{prob}$ constructs assigns a success probability p to the program σ_1 (and $1 - p$ to σ_2). To distinguish the two possible successor transitions the dummy actions $\mathit{tossHead}$ and $\mathit{tossTail}$ have been introduced. The statement $\mathbf{prob}$ is only for off-line projections. Obviously, it is never final.

Probabilistic Projections

With introducing the $prob$ statement it is possible to project over probabilistic programs. The idea is to find out what the probability is that a formula φ holds after execution of program σ. Therefore, a projection is started which ranges over all possible probabilistic traces. The overall probability is weighted over all possible traces. To access the value $PBel$ of the belief of the agent about φ when simulating σ, the fluent $pproj$ is introduced:

$$\begin{aligned}
&PBel(\varphi, \sigma, s) = p \equiv \\
&\quad \frac{\sum_{\{s', s^* | \varphi[s^*]\}} p(s', s) \cdot doPr(withPol(ll_{model}, \sigma), s', s^*)}{\sum_{s'} p(s', s)} = p.
\end{aligned}$$

In the above formula p is a binary functional fluent with the meaning that “in situation s the agent thinks that s' is possible with degree of likelihood $p(s, s')$”. This was introduced in the epistemic situation calculus (cf. (Reiter 2001)). $doPr$ was introduced in Chapter 3.3.4 on page 56 and ’returns’ the probability of a projection; in this case of the low-level model ll_{model} with program σ interleaved.

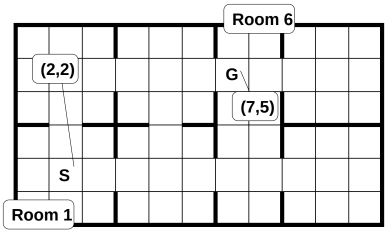


Figure 4.2: Maze66 from (Hauskrecht et al. 1998).

Procedures
As stated in the introduction of this section, we leave out the formal definition of procedures here. It demands a second-order definition of $trans$. The semantic of procedures is like those in common programming languages.

This concludes our definition of the language READYLOG w.r.t. all non-decision-theoretic extensions. These are given in the next section. Examples of applications of READYLOG are described in Chapters 5 and 6. Next, we concentrate on the decision-theoretic extensions of READYLOG.

4.2 On-line DT Planning with Passive Sensing

In Section 3.3.5 we introduced DTGOLOG. Essentially, the basic action theory together with the optimization theory defines an MDP which is solved applying a forward search value iteration algorithm. The DTGOLOG program is interpreted, choices in the program are resolved by choosing the optimal alternative w.r.t. the given optimization theory.

A DT Planning Example

In the following we illustrate how DTGOLOG calculates the optimal policy from a program. A robot should navigate from its start position S to a goal position G. It can perform one of the actions from the set $A_{base} = \{go\_right,\ go\_left,\ go\_up,\ go\_down\}$. Each of the actions bring the robot to one of its neighboring locations. The actions are stochastic, that is, a probability distribution over the effects of the action exists. Each action takes the agent to the intended field with probability of p, with probability $1 - p$ the robot will arrive at any other adjacent field. The maze shown is the well-known Maze66 domain from (Hauskrecht et al. 1998). In our example $p = 0.7$ which means that the action $right$ will succeed with probability 0.7, and with probability of 0.1 nature chooses one of the actions go_left, go_up, and go_down. The robot cannot go through the walls, if it tries, though, the effect is that it does not change its position at all. Figure 4.2 illustrates the scenario.

Accordingly, the basic action theory consists of the fluent loc, and the situation independent atoms $start$ and $goal$, and the stochastic actions go_right, go_left, go_up, and go_down. As these are stochastic actions we have to provide the predicates $choice(A, a, s)$ and $prob(n, a, s)$. This means that we have to define a set of deterministic actions r, l, u, d from which nature chooses when performing one of our navigation actions. For ease of exposition we assume that the outcomes for each action remain the same in each situation, i.e. $choice(go\_right) = choice(go\_left) = choice(go\_up) = choice(go\_down) \overset{def}{=} \{r, l, u, d\}$. The probability for each outcome is then $prob(r, go\_right) = prob(l, go\_left) = prob(u, go\_up) = prob(d, go\_down) \overset{def}{=} 0.7$ and $prob(n, A, s) \overset{def}{=} 0.1$ for the remaining action pairs.[2]

The successor state axiom for the location fluent then is defined as

$$\begin{aligned}
&loc(do(a,s)) = (x,y) \equiv \\
&\quad \exists x', y'.loc(s) = (x', y') \land \\
&\quad ((a = r \land x = x' + 1 \land y = y') \lor (a = l \land x = x' - 1 \land y = y') \lor \\
&\quad (a = u \land x = x' \land y = y' + 1) \lor (a = d \land x = x' \land y = y' - 1) \lor \\
&\quad (a \neq r \land a \neq l \land a \neq u \land a \neq d \land x = x' \land y = y')).
\end{aligned}$$

$start$ and $goal$ are situation independent and encode only the position of the start and the target position. In our example $start = (1, 1)$ and $goal = (7, 5)$. The reward function is defined as

$$reward(s) = \begin{cases} +1 & loc(s) = goal(s) \\ -1 & \text{otherwise} \end{cases}$$

To find the optimal path from S to G the robot is equipped with the program

```
proc navigate
  solve(while $\neg loc = goal$ do
    ($go\_right \mid go\_left \mid go\_up \mid go\_down$)
  endwhile, $h$)
endproc
```

DTGOLOG now interprets the program via the $BestDo$ predicates. As long as the robot is not at the goal location (and the horizon is not reached) DTGOLOG loops over the nondeterministic choice statement. At each iteration the interpreter expands a sub-tree for each of the actions inside the choice statement. As each of the actions are stochastic ones, for each outcome of each action the interpreter branches over the nature's choices again. This goes on until either the agent is located at the goal position or the horizon is reached. At the leaves of the computation tree over $BestDo$ (at the end of the recursion) the agent receives the reward for these final situations. Then, "going up" the computation tree for nondeterministic choices, the best alternative is evaluated and chosen for the policy. At nature's choices a conditional is added to the policy. With this

[2]Again, for ease of exposition, we do not distinguish between different partitions of the state space. As all probabilities have to sum up to 1 for each action, the probability mass of a stochastic action has to be redistributed over the possible outcomes. This means that at position $(1, 1)$ only the actions r and u are possible and thus the probability of the outcome for r is 0.875 and for the outcome u is 0.125

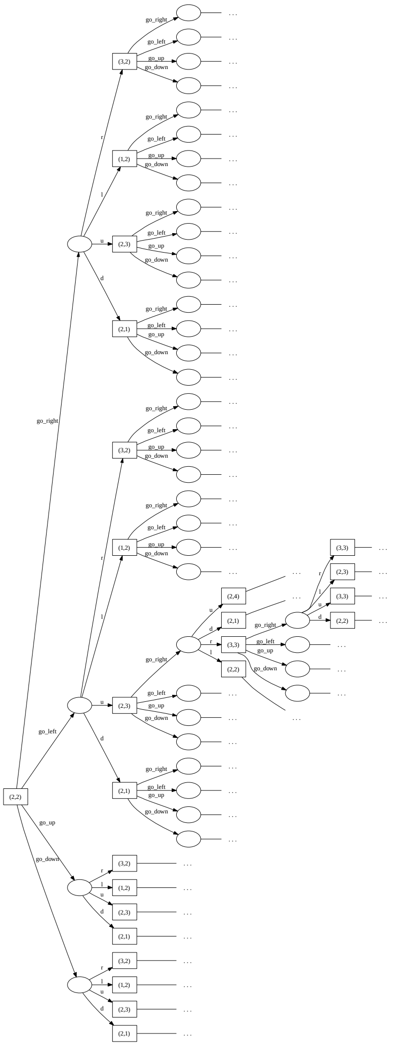


Figure 4.3: The Decision Tree for the Maze Example

```
u : senseEffect(u) : if(I_am_at(2, 1)) then
        |- u : senseEffect(u) : if(I_am_at(1, 1)) then
        |       |- u : senseEffect(u) : if(I_am_at(2, 1)) then  ...
        |- if(I_am_at(1, 2)) then
                |- r : senseEffect(e) : if(I_am_at(2, 2)) then
                |       |- u : senseEffect(u) : if(I_am_at(2, 1)) then
                |       |       |- u : senseEffect(u) : if(I_am_at(1, 1))
                |       |       |       |- u : senseEffect(u) : ...
```

Figure 4.4: The Optimal Policy for the Maze Domain

conditional, later, when the policy is executed, it offers an action for each possible outcome of the respective stochastic action. Coming up to the root node the computation terminates returning the policy, the value for the policy, and its probability of success. Part of the computation tree is shown in Figure 4.3. The resulting policy is depicted in Figure 4.4.

Note that with the $BestDo$ computation of nondeterministic choice and stochastic actions

$$
\begin{aligned}
&BestDo((p_1|p_2); p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \ldots \\
&\quad ((v_1, p_1) \geq (v_2, p_2) \wedge \pi = \pi_1 \wedge pr = pr_1 \wedge v = v_1) \vee \\
&\quad (v_1, p_1) < (v_2, p_2) \wedge \pi = \pi_2 \wedge pr = pr_2 \wedge v = v_2)
\end{aligned}
$$

$$
\begin{aligned}
&BestDoAux(\{n_1, \ldots, n_k\}, p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \ldots \\
&\quad v = v' + v_1 \cdot prob(n_1, s) \wedge pr = pr' + p_1 \cdot prob(n_1, s)
\end{aligned}
$$

(Item 4 and 6 on page 60) and given here again the agent computes a policy following an expected reward criterion as it weights the value according to its success probabilities, i.e.

$$V(\pi) = \max \mathrm{E}\left(\sum_{h=0}^{H} R(s^h|\pi)\right).$$

The robot is now endowed with a conditional program which tells it, for each of the positions it can reach, which is the best action to advance to the goal position. One has to remark that the policy only yields an action for the projected locations the robot reached during planning, and up to the given fixed horizon. It does not have any idea which action to take at location, say $(10, 4)$. This might at first seem to be a disadvantage, but on second thoughts this turns out to be one of the advantages of DTGOLOG. With standard solution techniques to MDPs like value iteration one would have a solution for each of the locations, but for the cost that value iteration has to iterate several times over all states. With DTGOLOG this can be avoided by only expanding the reachable successor locations.

4.2.1 Discussion: Soutchanski's On-line DTGolog

The strength of the forward search value iteration algorithm used in DTGOLOG is that the structure of the MDP is induced while interpreting the DTGOLOG program. The successor state axiom for a fluent induces the transition between states of the MDP. The next state of the MDP is the one that is accessible from the current situation by executing the current action. Thus, only the accessible situations are a successor state in the induced MDP. The advantage is that the state space of the MDP does not need to be known explicitly in advance, it is implicitly defined by the basic action theory.

The original version of DTGOLOG operates in an off-line mode, that is, it first computes a policy for the whole program and only then initiates execution. As was observed already in (De Giacomo and Levesque 1999), this is not practical for large programs and certainly not for applications with tight real-time constraints such as robotic soccer. In the extreme one would only want to reason about the next action of a program, execute it and then continue with the rest of the program. This is the basic idea of an on-line interpretation of a GOLOG program (De Giacomo and Levesque 1999) as described in Chapter 3.3.3.

A nice feature of on-line interpretation is that the step-wise execution of a program can easily be interleaved with other exogenous actions or events, which are supplied from outside.

With the basic transition mechanism in hand, it is in principle, not hard to reintroduce off-line reasoning for parts of the program. In the case of DTGOLOG, Soutchanski proposed for that purpose an interleaving of off-line planning and on-line execution. We show an excerpt of his interpreter implemented in Prolog. We only consider the case of executing deterministic and sensing actions, leaving out stochastic actions:

```
online(E, S, H, Pol, U) :-
    incrBestDo(E, S, ER, H, Pol1, U1, Prob1),
    (final(ER, S, H, Pol1, U1), Pol = Pol1, U = U1;
      reward(R, S), Pol1 = (A : Rest),
      (agentAction(A), doReally(A), !,    %% deterministic action
      online(ER, do(A, S), H, PolFut, UFut),
      Pol = (A : PolFut), UisR + UFut;
      senseAction(A), doReally(A), !,    %% sensing action
      online(ER, do(A, S), H, PolFut, UFut),
      Pol = (A : PolFut), UisR + UFut;
    ...
      )
    ).
```

Basically, the interpreter *online* calculates a policy π for a given program e up to a given horizon h, executes its first action (*doReally*(a)) and recursively calls the interpreter with the remaining program again.

To control the search while optimizing Soutchanski proposes an operator *optimize* defined by the following macro:

$$\begin{aligned}
&\mathit{IncrBestDo}(\mathit{optimize}(p_1); p_2, s, p_r, h, \pi, u, pr) \overset{def}{=} \\
&\quad \exists p'.\mathit{IncrBestDo}(p_1; \mathit{Nil}, s, p', h, \pi, u, pr) \wedge \\
&\quad (p' \neq \mathit{Nil} \wedge p_r = (\mathit{optimize}(p'); p_2) \vee \\
&\quad p' = \mathit{Nil} \wedge p_r = p_2).
\end{aligned}$$

This has the effect that p_1 is optimized and the resulting policy is executed before p_2 is even considered. It has the advantage that the user can deal with explicit (active) sensing actions by restricting $\mathit{optimize}$ to never go beyond the next sensing action. Note that the predicate $\mathit{IncrBestDo}$ does the same as the BestDo, namely calculating the policy for the given program.

Nevertheless the approach has a number of shortcomings. First note that, in the definition of the interpreter online, after executing only one action of a computed policy, the optimizer is called again. This means that large parts of the program are re-optimized over and over again, which is computationally too expensive for the kind of real-time decision making which we have in mind. While it would not be that difficult to modify the interpreter so that a complete policy is executed before computing the next one, the main problem has to do with sensing. Soutchanski's approach only addresses active sensing, where it is reasonable to assume that there usually are a number of non-sensing actions happening between two sensing actions. With passive sensing, this might not be the case. As shown in (Grosskreutz and Lakemeyer 2001), one way to deal with this kind of ubiquitous sensing is to keep it in the background and not even refer to it in the robot control program. Adopting this idea for DTGOLOG has two consequences. For one, policy generation needs to work with a suitable model of how the world evolves. For another, during policy execution it needs to be monitored whether the way the world actually evolves is compatible with the model assumptions. This is what Soutchanski and, for that matter, the original DTGOLOG neglected to do. As an indication of this note that when Soutchanski's interpreter executes an action a from the policy $(\mathit{doReally}(a))$, it is tacitly assumed that a is still executable, which may not be the case. Perhaps more seriously, the whole policy may become obsolete due to unforeseen developments in the world, and this needs to be detected as well.

4.2.2 Extending Passive Sensing

To deal with incomplete knowledge about the environment, GOLOG was extended with sensing actions (Lakemeyer 1999; De Giacomo and Levesque 1999). These special actions allow an agent to query its sensors so as to gather information about the environment. This approach has, under certain circumstances, several drawbacks. When sensor values must be updated very frequently, acquiring world information through sensing actions is not feasible. The agent is busy with executing sensing actions most of the time. Another problem occurs when off-line planning is interleaved with on-line execution. If the plan relies on the on-line information, the result of planning might be inconsistent due to wrong sensing results. It is generally not possible to plan ahead of sensing actions when deploying an active sensing approach. As we discussed in the previous section this is the main disadvantage of Soutchanski's on-line DTGOLOG approach.

What is needed is a passive sensing approach which performs updates of the sensor values in the background. Proposals for passive sensing approaches are e.g. (Poole 1997; Grosskreutz and Lakemeyer 2001).

Our approach to passive sensing is based on (Grosskreutz and Lakemeyer 2001). They propose a system architecture where the high-level controller starts low-level processes via a so-called action register. (We already discussed this issue; nevertheless we recap their system architecture for a coherent presentation of the matter here). To initiate an action the high-level controller sends a command to the register. This command is passed through to the execution module which in turn takes care for executing of the action in the real world. From now on, the high-level controller is no longer concerned with monitoring the execution of the action. This is done asynchronously by the execution layer. When the execution of the action is finalized, the low-level control indicates this by sending a reply message to the high-level controller via the action register. The high-level controller can now react on this specific message. Between a $send$ and a $reply$ message the high-level controller is free for other tasks. The communication between high-level and low-level system is realized through a special fluent $register$ and the two actions $send$ and $reply$. The high-level controller invokes the low-level process with settling a $send$ action, the low-level process answers with a $reply$ message when the low-level process is finished (This becomes more evident in Chapter 4.4 when discussing the implementation of $send$ and $reply$).

With this form of communication one has to distinguish between two modes: The on-line execution and the off-line projection task. During on-line execution the fluent $register$ simply blocks until the low-level system has executed the command in the real world. Making use of prioritized concurrency the high-level controller can go on with its program. When projecting, a model of the low-level process is executed. The $register$ fluent is also used to synchronize sensor updates in the background. With a special $ccUpdate$ action an exogenous event is raised which does the sensor update. In the off-line mode the fluents are updated according to a low-level process of the currently executed action, during on-line execution the real sensor value is used.

When dealing with a large amount of world-model information like in robotic soccer the integration of the update action takes longer than the decision cycle of the agent. Updating a complete world model in the simulated soccer domain, for example, takes more than $100\ ms$ which is longer than the decision cycle in the SIMULATION LEAGUE (cf. Chapter 5.2). To deal with this problem we extended the system architecture proposed in (Grosskreutz and Lakemeyer 2001) by an explicit world model. To access the data stored in the world model we introduce a new fluent type, the so-called *on-line fluent*. The difference to ordinary fluents is that they are directly connected to the world model. They are initialized with the value $undefined$. With a special action $exogfUpdate$[3] one can update the respective fluents. When the action $exogfUpdate$ is executed a copy of the real sensor data at that specific moment in time is copied to the internal world model of the READYLOG agent. In Figure 4.5 we present the extended system architecture. As in (Grosskreutz and Lakemeyer 2001) we use the special $register$ fluent and the message passing between high-level and low-level control. The presence of an explicit world model extends the architecture. The special action $exogfUpdate$ which is sent via the register initiates an update of the world model of the agent. The shaded arc "world" denotes the connection to the real world. One has to think of this as a rather complex layered system architecture which has access to the actuators and can gather data from the sensors.[4] What is needed is that information taken from the real world can

[3]The idea behind this action is very similar to the $ccUpdate$ action proposed in (Grosskreutz and Lakemeyer 2001). The difference is how this action is handled by the READYLOG interpreter (see Chapter 4.4).

[4]In Chapter 6 we show such a system when introducing our robot control software.

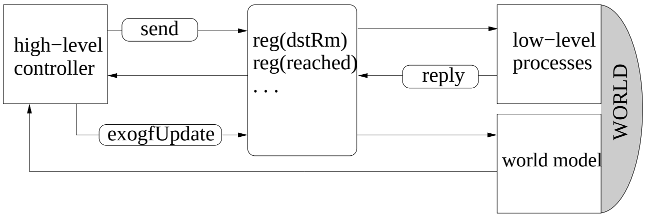


Figure 4.5: The Extended Architecture

be updated asynchronously on demand.[5] The possibility to asynchronously update sensor values allows for on-line passive sensing. In the on-line execution mode of the interpreter, sensor updates arrive synchronously, or when they are explicitly triggered with an $\mathit{exogfUpdate}$ action. Note that there are no mechanisms provided to check consistency of the world model information. In the domain implementation one has to check for consistency. In the off-line projection mode the last world state is taken and projected with the low-level processes. This simple method provides the interpreter with an efficient form of a passive sensing facility.

With this form of on-line passive sensing regression of on-line fluents is no longer possible. Regression is the process of reducing a formula with a complex situation term to a formula mentioning only S_0 for sort situation. But with on-line fluents which are updated with each occurrence of the action $\mathit{exogfUpdate}$ it is not possible to regress an action history over an occurrence of $\mathit{exogfUpdate}$. Consider the following example. Let $f_1(do(a,s)) = v \equiv a = a_1 \wedge v = f_2(s) + 1 \vee a \neq a_1 \wedge v = f_1(s)$ be the successor state axiom of the fluent f_1 with $f_1(S_0) = 0$, and let f_2 be an on-line fluent which is updated by the $\mathit{exogfUpdate}$ action to take the value 1, i.e. $f_2(do([\mathit{exogfUpdate}], s0)) = 1$. Now suppose the action a_1 has been performed. Then, $f_1(do([a_1, \mathit{exogfUpdate}], s0)) = 2$. If another update action changes the fluent value of f_2 to, say 0, the result of regressing the fluent value for f_1 should be be 1 but in fact the value of f_1 is still 2. The reason is that not the whole world model is stored for each update action. It takes only the reference to the last world model update. So, care has to be taken when using on-line fluents. When only on-line fluents are used the problem does not exist when fluents are not regressed further as the last update action. This yields a fast method to access values for on-line fluents, but in general correctness cannot be guaranteed. To solve this problem, we have to use progression instead. With progression a new initial database is computed. Progression is also indispensable for long-time operations of a READYLOG interpreter and we discuss our approach to progression in Chapter 4.4. So why does progression solve our problem here? When a new initial database is calculated each time a world model update action is performed one can guarantee that no fluent refers to an old on-line fluent value. For off-line projections or decision-theoretic planning which makes use of regression to access fluent values this problem does not have an impact. The $\mathit{exogfUpdate}$ action is an exogenous action which is ignored during projections anyway. This problem only occurs when executing programs or policies on-line. We will discuss how exogenous actions are integrated when introducing the implementation of the Readylog interpreter in Chapter 4.4.

[5]By 'on demand' we mean that this feature can be turned off, for example in projections of READYLOG programs.

4.2.3 Execution Monitoring for Policies

As we have seen in the discussion of Soutchanski's approach re-optimization of the remaining program is generally not feasible in real-time domains.[6] Our first modification of on-line DTGOLOG is to make sure that the whole policy and not just the first action is executed. For this purpose we introduce the operator $solve(p, h)$ for a program p and a fixed horizon h.

$$\begin{aligned}&trans(solve(p,h), s, \delta, s') = \\ &\quad \mathbf{if}\ \exists \pi, v, pr.BestDo(p, s, h, \pi, v, pr) \wedge\ \delta = applyPol(\pi) \wedge s' = s\ \mathbf{then}\ 1 \\ &\quad \mathbf{else}\ 0\end{aligned}$$

The predicate $BestDo$ first calculates the policy for the whole program p. For now the reader may assume the definition of Chapter 3.3.5, but we will see below that it needs to be modified. This policy is then scheduled for execution as the remaining program. However, as discussed before, the policy is generated using an abstract model of the world to avoid sensing, and we need to monitor whether π remains valid during execution. To allow for this special treatment, we use the special construct $applyPol$, which is defined below. Note that the $solve$ statement never reaches a final configuration as further transitions are needed to execute the calculated policy.

$$Final(solve(p,h), s, \delta', s') \equiv false$$

To see why we need to modify the original definition of $BestDo$ and, for that matter, the one used by Soutchanski, we need to consider, in a little more detail, the idea of using a model of the world when planning vs. using sensor data during execution. The following fragment of the control program of our soccer robots might help to illustrate the problem:

```
$\mathbf{while}\ game\_on\ \mathbf{do} \ldots;$
    $solve(\ldots;$
        $\mathbf{if}\ \exists x, y(ball\_pos(x,y) \wedge reachable(x,y))$
        $\mathbf{then}\ intercept$
        $\mathbf{else} \ldots; \ldots\ , h)$
$\mathbf{endwhile}$
```

While the game is still on, the robots execute a loop where they determine an optimal policy for the next few (typically less than six) actions, execute the policy and then continue the loop. One of the choices is intercepting the ball which requires that the ball is reachable, which can be defined as a clear trajectory between the robot and the ball. Now suppose $BestDo$ determines that the if-condition is true and that $intercept$ has the highest utility. In that case, since $intercept$ is a stochastic action, the resulting policy π contains $\ldots intercept; senseEffect(intercept); \ldots$. Note, in particular, that the if-condition of the original program is not part of the policy. And this is where the problem lies. For during execution of the policy it may well be the case that the ball is no longer reachable because an opponent is blocking the way. In that case $intercept$ will fail and it makes sense to abort the policy and start planning for the next moves. Therefore, the if-condition should be re-evaluated using the most up-to-date information about the world provided

[6] But see, for example, (Beetz et al. 2004) for an approach to re-optimization using learning techniques.

by the sensors and compared to the old value. Hence we need to make sure that the if-condition and the old truth value are remembered in the policy.

This means we need to modify the definition of $BestDo$ for those cases involving the evaluation of logical formulas. First, we consider conditionals.

$$\begin{aligned}
&BestDo(\mathbf{if}\ \varphi\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{endif}; p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \varphi[s] \wedge \exists \pi_1 . BestDo(p_1; p, s, h, \pi_1, pr) \wedge \\
&\qquad \pi = \mathfrak{M}(\varphi, true); \pi_1 \vee \\
&\quad \neg\varphi[s] \wedge \exists \pi_2 . BestDo(p_2; p, s, h, \pi_2, v, pr) \wedge \\
&\qquad \pi = \mathfrak{M}(\varphi, false); \pi_2
\end{aligned}$$

While-loops are treated in a similar way:

$$\begin{aligned}
&BestDo(\mathbf{while}\ \varphi\ \mathbf{do}\ p'\ \mathbf{endwhile}; p, s, h, \pi, v, pr \overset{def}{=} \\
&\quad \varphi[s] \wedge \exists \pi_1 . BestDo(p'; while(\varphi, p'); p, s, h, \pi_1, v, pr) \\
&\quad \wedge \pi = \mathfrak{M}(\varphi, true); \pi_1 \vee \\
&\quad \neg\varphi[s] \wedge \exists \pi_2 . BestDo(p; s, h, \pi_2, v_2, pr_2) \\
&\quad \wedge \pi = \mathfrak{M}(\varphi, false); \pi_2
\end{aligned}$$

The only difference compared to the original $BestDo$ is that we prefix the generated policy with a marker $\mathfrak{M}(\varphi, true)$ in case φ turned out to be true in s and $\mathfrak{M}(\varphi, false)$ if it is false. The treatment of a test action $\varphi?$ is even simpler, since only the case where φ is true matters. If φ is false, the current branch of the policy is terminated, which is indicated by the $Stop$ action.

$$\begin{aligned}
&BestDo(\varphi?; p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \varphi[s] \wedge \exists \pi' . BestDo(p, s, h, \pi', v, pr) \wedge \\
&\quad \pi = \mathfrak{M}(\varphi, true); \pi' \vee \\
&\quad \neg\varphi[s] \wedge \pi = Stop \wedge pr = 0 \wedge v = reward(s)
\end{aligned}$$

Next, we will show how our annotations will allow us to check at execution time whether the truth value of conditions in the program at planning time are still the same and what to do about it when they are not. Before that, however, it should be mentioned that explicit tests are not the only reason for a possible mismatch between planning and execution. To see that note that when a primitive action is entered into a policy, the resp. $BestDo$ predicate checks its precondition. Of course, it could happen that the same action is no longer possible at execution time. It turns out that this case can be handled without any special annotation.

Now that we have modified $BestDo$ so that we can discover problems at execution time, all that is left to do is to define the actual execution of a policy. Given our initial definition of $trans(solve(p, h), s, \delta, s')$, this means that we need to define $trans$ for the different cases of $applyPol(\pi)$. To keep the definitions simple, let us assume that every branch of a policy ends with

$Stop$ or Nil, where Nil represents the empty program.

$$\begin{aligned}&\mathit{trans}(\mathit{applyPol}(Nil), s, \delta, s')\\ &\qquad \mathbf{if}\ s = s' \wedge \delta = Nil\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\ &\mathit{trans}(\mathit{applyPol}(\mathit{Stop}), s, \delta, s') =\\ &\qquad \mathbf{if}\ s = s' \wedge \delta = Nil\ \mathbf{then}\ 1\ \mathbf{else}\ 0\end{aligned}$$

Given the fact that configurations with Nil as the program are always final, i.e. execution may legally terminate, this simply means that nothing needs to be done after $Stop$ or Nil.

In case a marker was inserted into the policy we have to check that the test performed at planning time still yields the same result. If this is the case we are happy and continue executing the policy, that is, $\mathit{applyPol}$ remains in effect in the successor configuration. But what should we do if the test turns out different? We have chosen to simply abort the policy, i.e. the successor configuration has Nil as its program. While this may seem simplistic, it seems the right approach for applications like robotic soccer. For example, consider the case of an intercept. If we find out that the path is blocked, the intercept will likely fail and all subsequent actions in the policy become meaningless. Moreover, a quick abort will enable immediate re-planning according to the control program, which is not a bad idea under the circumstances. Another possibility to handle policies which have become invalid is to try to repair them. In general, repairing a policy is computationally as demanding as planning from scratch. Under certain circumstances re-planning may be to be possible. We discuss this issue in Chapter 8 as future work.

$$\begin{aligned}&\mathit{trans}(\mathit{applyPol}(\mathfrak{M}(\varphi, v); \pi), s, \delta, s') =\\ &\qquad \mathbf{if}\ s = s' \wedge (v = \mathit{true} \wedge \varphi[s] \wedge \delta = \mathit{applyPol}(\pi) \vee\\ &\qquad v = \mathit{false} \wedge \neg\varphi[s] \wedge \delta = \mathit{applyPol}(\pi) \vee\\ &\qquad v = \mathit{true} \wedge \neg\varphi[s] \wedge \delta = Nil \vee v = \mathit{false} \wedge \varphi[s] \wedge \delta = Nil)\ \mathbf{then}\ 1\ \mathbf{else}\ 0\end{aligned}$$

If the next construct in the policy is a primitive action other than a stochastic action or a $\mathit{senseEffect}$, then we execute the action and continue executing the rest of the policy. As discussed above, due to changes in the world it may be the case that a has become impossible to execute. In this case we again abort the rest of the policy with the successor configuration $\langle Nil, s\rangle$.

$$\begin{aligned}&\mathit{trans}(\mathit{applyPol}(a; \pi), s, \delta, s') =\\ &\qquad \mathbf{if}\ \exists\delta'.\mathit{trans}(a; \pi, s, \delta', s') > 0 \wedge \delta' = \mathit{applyPol}(\delta') \vee\\ &\qquad \neg\mathit{Poss}(a[s], s) \wedge \delta = Nil \wedge s' = s\ \mathbf{then}\ 1\ \mathbf{else}\ 0\end{aligned}$$

If a is a stochastic action as presented in Chapter 3.3.5, we obtain

$$\begin{aligned}&\mathit{trans}(\mathit{applyPol}(a; \mathit{senseEffect}(a); \pi), s, \delta, s') =\\ &\qquad \mathbf{if}\ \exists\delta'.\mathit{trans}(\mathit{senseEffect}(a); \pi, s, \delta', s') > 0 \wedge \delta = \mathit{applyPol}(\delta'))\ \mathbf{then}\ 1\ \mathbf{else}\ 0\end{aligned}$$

Note the subtlety that a is ignored by trans. This has to do with the fact that stochastic actions have no direct effects according to the way they are modeled in DTGOLOG. Instead one needs to perform $\mathit{senseEffect}$ to find out about the actual effects. Therefore $\mathit{senseEffect}$ is handled separately as case (3) of the icpxeq predicate in our interpreter (cf. Chapter 4.4). The action a is executed ($\mathit{execute}(\mathit{Act})$). Then the interpreter waits until new sensor updates arrive ($\mathit{wait\_for\_next\_update}$) to determine the outcome of action a which was chosen by nature.

Finally, if we encounter an if-construct, which was inserted into the policy due to a stochastic action, we determine which branch of the policy to choose and go on with the execution of that branch.

$$\begin{aligned}&trans(applyPol(\mathbf{if}\ \varphi\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2\ \mathbf{endif}), s, \delta, s') = \\ &\quad \mathbf{if}\ \varphi[s] \land trans(applyPol(\pi_1), s, \delta, s') > 0 \lor \\ &\quad \neg\varphi[s] \land trans(applyPol(\pi_2), s, \delta, s') > 0\ \mathbf{then}\ 1\ \mathbf{else}\ 0\end{aligned}$$

If we reach the horizon we have to stop the execution of the policy, which, if nothing went wrong, has reached a final configuration by then.

$$Final(applyPol(p, h), s) \equiv Final(p, s) \lor h = 0$$

With these definitions we are able to detect when a policy becomes invalid during execution. As stated above we currently handle invalid policies by simply invoke re-planning. Other possibilities to handle this are left to future work.

4.2.4 Extending Stochastic Action Models

The decision-theoretic extension of GOLOG makes use of a notion of stochastic actions as introduced in Chapter 3.3.5. A stochastic action is described by two deterministic actions, one referring to the name of the stochastic action with no effect, and a number of deterministic actions denoting the outcomes of the stochastic action. Formally, the predicate $choice(A, a, s) \overset{def}{=} \psi_1 \supset (a = N_1^1 \lor \cdots \lor a = N_1^n) \land \cdots \land \psi_m \supset (a = N_m^1 \lor \cdots \lor a = N_m^n)$ defines the different outcomes $N_1, \ldots, N_k$ of the stochastic action A. For ease of presentation we use the simplified, situation-independent version $choice'(A) \overset{def}{=} \{N_1, \ldots, N_k\}$, as defined in (Soutchanski 2003). Associated with this is a probability of the occurrence of the respective outcome, defined by a set of predicates $prob(N_i, A, s) = p_i$ with p_i a probability, one for each outcome. Finally, one needs a predicate $senseCond(N_i, \varphi)$ which states that the outcome N_i occurred if φ is true.

An outcome is described by a primitive action whose effects changes the respective fluents. Consider the example of a stochastic action with n outcomes. For each of these outcomes one distinct primitive action has to be modeled in the domain description. Further, for each outcome action which changes a fluent value a separate case in the successor state axiom of that fluent has to be modeled. In larger domain descriptions with many fluents and many stochastic actions, many different outcome actions have to be modeled and many cases in the definition of successor state axioms have to be regarded. An easier way to define an outcome of an action is to simply state how this outcome changes a fluent value without the need to define a new primitive action. With a simple action $set(f, v)$ which sets fluent f to respective values v the tasks could be done.[7] Thus, one can avoid a separate case for each outcome in the successor state axiom of the respective fluent which is changed. If more than one fluent should be changed, one must allow sequences of *set* actions. This means that an outcome then can be described by the action sequence $set(f_1, v_1); set(f_2, v_2); \ldots; set(f_n, v_n)$. The effect is the same as encoding this sequence by a single primitive action. The advantage is, though, that the implementor can keep the overview which

[7]Note that f refers to the reified version of the fluent f (cf. Section 4.1.2). This allows to take the fluent term as argument of an action term.

fluents are changed without the need to remember all outcome actions. While this is an argument from a practical point of view (which is nevertheless important when dealing with large domain descriptions), it also means that in successor state axioms less cases have to be distinguished. The drawback, on the other side, is that we restrict ourselves to finite domains. Implicit effects cannot expressed with this form of setting fluent values. For example, we cannot implicitly formulate that everything is broken in the vicinity of an exploding bomb. Instead, we must make all effects explicit. But for practical applications this limitation did not turn out to be too restrictive.

We now alter the notion of a stochastic action to so-called stochastic procedure models in this spirit. The idea is similar to the idea of using low-level processes to describe a model of a real-world process and making use of the $prob$ statement to describe probabilistic programs. We want to stress again that the reason for introducing these special kind of model is mainly that with the notion of stochastic models the encoding of the outcomes of a stochastic action for large domain descriptions is simplified. In the following we define stochastic procedures and show that they are at least as expressive as the stochastic actions introduced in Chapter 3.3.5.

A stochastic model $\mathbf{procmodel}$ is a special procedure which encodes the outcomes of a stochastic action. An example for a stochastic procedure is

$\mathbf{procmodel}\ A$
  $a;\ b;\ \varphi?;\ c;\ \mathbf{if}\ \psi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endif};$
  $sprob(\{(n_{success}), p_1, \varphi_1), (n_{failure}, 1 - p_1, \varphi_2)\})$
$\mathbf{endprocmodel}$

We refer to the body of a stochastic procedure as σ, the part before the $sprob$ statement we refer to as the preamble ϱ of σ. So, the effect of a stochastic action A can be defined by a program which describes the outcomes of the stochastic action in terms of σ. In the above example we define a success case which is chosen by nature with probability p_1, and a failure case with the counter probability of p_1. The φ's are used to distinguish which case occurred in reality later, when the action is executed. The φ's have to be mutually exclusive to distinguish the different cases.[8] A stochastic action can be modeled using restricted READYLOG programs which are defined as follows.

Definition 3 *A* READYLOG *program σ is called restricted when the following holds:*

1. *$\sigma = nil$ (the empty program) is restricted;*
2. *$\sigma = \varphi?$ is restricted;*
3. *$\sigma = a$ with a being a primitive action is restricted;*
4. *if σ_1 and σ_2 are restricted then $\sigma = \mathbf{if}\ \varphi\ \mathbf{then}\ \sigma_1\ \mathbf{else}\ \sigma_2\ \mathbf{endif}$ is restricted;*
5. *if σ_1 and σ_2 are restricted then $\sigma = \sigma_1; \sigma_2$ is restricted;*
6. *if σ' is restricted and does not mention $sprob$ then $\sigma = \sigma'; sprob(\{(n_1, p_1, \phi_1), \ldots, (n_k, p_k, \phi_k)\})$ where $n_1, \ldots, n_k$ are restricted programs without $sprob$, $p_1, \ldots, p_k \in [0, 1]$ and $\sum_i p_i = 1$.*

[8]Note that the same applies with Reiter's approach to stochastic actions.

A stochastic action model may only mention tests, primitive actions, conditionals, sequences, and a special statement $sprob$. $sprob$ corresponds to the **prob** statement known from pGOLOG (cf. Chapter 3.3.4). The effect of the action can be modeled by a restricted READYLOG program. The $sprob$ statement, basically, defines the same as the predicates $choice$, $prob$, and $senseCond$, encoding a description how the world changes by an outcome chosen by nature, the probability that nature will choose it, and a condition with which the agent is able to sense which of the possible outcomes were chosen. The difference to $choice$ is that instead of referring to an action term, we describe the effects directly by a restricted program.

For the case of a stochastic action we have to change the respective $BestDo$ predicate in our semantic definition.

$$\begin{aligned}
&BestDo(A; p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \exists P.procmodel(A, P) \land Poss(A, s) \land \\
&\quad\quad \exists \pi', v', pr'.BestDoAux(P, p, s, h-1, \pi', v', pr') \\
&\quad\quad \pi = a; \pi' \land v = reward(s) + v' \land pr = pr' \lor \\
&\quad \neg Poss(a, s) \land \pi = Stop \land v = 0 \land pr = 0
\end{aligned} \tag{4.1}$$

From the definition of the procedure model for the stochastic action A we take the procedure body and process it with a predicate $BestDoAux$. For primitive actions, test, and conditionals the definition of $BestDoAux$ is the same as for $BestDo$. We therefore omit it here. The interesting case is the $sprob$ statement which finally encodes the stochastic outcomes.

$$\begin{aligned}
&BestDoAux(sprob(\{(n_1, p_1, \varphi_1), \ldots, (n_k, p_k, \varphi_k)\}, p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \exists \delta, s'.trans^*(n_1, s, \delta, s') > 0 \land Final(\delta, s') \land \\
&\quad\quad \exists \pi', v', pr'.BestDo(p, s', h, \pi', v', pr') \land \\
&\quad\quad \exists \pi'', v'', pr''.BestDoAux(\{(n_2, p_2, \varphi_2), \ldots, (n_k, p_k, \varphi_k)\}, p, s, h, \pi'', v'', pr'') \land \\
&\quad\quad \pi = \textbf{if } \varphi_1 \textbf{ then } \pi' \textbf{ else } \pi'' \textbf{ endif} \land \\
&\quad\quad v = v'' + p_1 \cdot v' \land pr = pr'' + p_1 \cdot pr' \lor \\
&\quad \forall \delta, s'.\neg(trans^*(n_1, s, \delta, s') > 0 \land Final(\delta, s')) \land \\
&\quad\quad BestDoAux(\{(n_2, p_2, \varphi_2), \ldots, (n_k, p_k, \varphi_k)\}, p, s, h, \pi, v, pr)
\end{aligned}$$

$$\begin{aligned}
&BestDoAux(sprob(\{(n_1, p_1, \varphi_1)\}), p, s, h, \pi, v, pr) \overset{def}{=} \\
&\quad \exists \delta, s'.trans^*(n_1, s, \delta, s') > 0 \land Final(\delta, s') \land \\
&\quad\quad \exists \pi', v', pr'.BestDo(p, s', h, \pi', v', pr') \land \pi = ?\varphi_1; pi' \land v = v' \cdot p_1 \land pr = pr' \cdot p_1 \lor \\
&\quad \neg(\exists \delta, s'.trans^*(n_1, s, \delta, s') > 0 \land Final(\delta, s')) \land \\
&\quad\quad \pi = Stop \land v = 0 \land pr = 0
\end{aligned}$$

We are interested in the appropriate continuation of the remaining policy in the situation s'. s' is the situation which describes the world after the projection of n_1. The projection can be done with the predicate $trans^*$ as n mentions only primitive (deterministic) actions, tests, and conditionals. The remaining policy is computed with $BestDo$ over program p (the rest program) in situation s'. Similar to DTGOLOG we have, for each outcome mentioned by $sprob$, to introduce a conditional into the policy π. The value of the remaining policy is appropriately weighted with the probability p_i of the occurrence of the outcome n_i.

So what do we obtain from this form of stochastic action models? The implementation of models is more intuitive for the implementor, she does not have to model many primitive actions which describe the effect of an outcome, and she does not need to keep them in mind when modeling a stochastic action. This is an advantage for the practitioner implementing large domains. We give an example of such a stochastic procedure model on page 178 in Chapter 6.6 where we describe an "intercept ball" action in the context of the robotic soccer domain.

The next question concerns correctness of our approach w.r.t. the policies calculated with our stochastic action models in comparison to the original approach used in DTGOLOG. If, our new approach calculates the same policies for the same stochastic actions, then it calculates the correct result. First note the difference in the semantic definition of $sprob$ and $choice$. In the original approach the outcomes of a stochastic action were modeled by primitive actions. Our extension, though, allows for restricted programs as models for the outcomes of an stochastic action. What we show in the following is that the new $sprob$ statement when applied to primitive actions yields the same policies than the original approach to stochastic actions in DTGOLOG, and thus our approach yields correct results. We further show that each restricted READYLOG program can be expressed by a new program consisting of a primitive action only, which together with a new basic action theory models the effect of the restricted program. We start with some Lemmas showing that sequences of primitive actions, test action, and conditionals, can be simulated by primitive actions. The results of these are summarized in Theorem 2. Finally, we show that both semantics calculate the same policies. But first, we need the following theorem form Pirri and Reiter which states that the regression operator does not change the validity of formula.

Theorem 1 (Pirri and Reiter 1999) *Let $\mathcal{D}$ be a basic action theory, α a situation calculus formula, and $\mathcal{R}$ be the regression operator introduced in Definition 2 on page 44. Then, $\mathcal{D} \models \alpha \equiv \mathcal{R}[\alpha]$.*

Now, we are able to show that the effects of an arbitrary long sequence of actions can also be imparted to the world by a single primitive action. Of course, the new action may only be possible iff the original sequence is possible. Therefore, we also have to care for the precondition axioms of the action. In an earlier paper, Gu (2003) used the same idea to model stochastic macro actions. In her paper though, the claim that a sequence of actions can be simulated by a single primitive action was not proved. Moreover, from the proof below we yield a construction method for a new basic action theory $\mathcal{D}'$ which entails our new action.

Lemma 1 *Let $\sigma = a_1; \ldots ; a_n$ be a sequence of primitive actions. Let $\mathcal{D}$ be a basic action theory with $\mathcal{D}_{ap}$ the set of precondition axioms and $\mathcal{D}_{ssa}$ the set of successor state axioms. Let α be a new action with the precondition axiom $Poss(\alpha, s) \equiv \Pi_\alpha$ with*

$$\Pi_\alpha \stackrel{def}{=} \mathcal{R}\left[Poss(a_1, s) \land \left(\bigwedge_{i=2}^{n} Poss\left(a_i, do\left([a_1; \cdots ; a_{i-1}], s\right)\right) \right) \right], \tag{4.2}$$

$\mathcal{D}_{ap}^{\alpha} = \mathcal{D}_{ap} \cup \{\Pi_\alpha\}$ be the new set of precondition axiom for action α, and the effects of action α

are such that the set $\mathcal{D}^{\alpha}_{ssa}$ of successor state axioms for fluents F can be derived as:

$$F(\vec{x}, do(a,s)) \equiv (a \neq \alpha) \wedge [F(\vec{x}, s) \wedge \neg\gamma^{-}_{F}(\vec{x}, a, s) \vee \gamma^{+}_{F}(\vec{x}, a, s)] \vee (a = \alpha) \wedge \Phi^{\sigma}_{F}$$

with

$$\Phi^{\sigma}_{F} \overset{def}{=} \mathcal{R}\Bigg[F(\vec{x}, s) \wedge \bigwedge_{i=1}^{n} \neg\gamma^{-}_{F}(\vec{x}, a_i, do([a_1, \ldots, a_{i-1}], s)) \vee \tag{4.3}$$
$$\bigvee_{i=1}^{n} \left(\gamma^{+}_{F}(\vec{x}, a_i, do([a_1, \ldots, a_{i-1}], s) \wedge \bigwedge_{j=i+1}^{n} \neg\gamma^{-}_{F}(\vec{x}, a_j, do([a_1, \ldots, a_{j-1}], s))\right)\Bigg]$$

Let $\mathcal{D}^{\alpha} = \mathcal{D} \cup \mathcal{D}^{\alpha}_{ap} \cup \mathcal{D}^{\alpha}_{ssa}$, then,

1. *$\mathcal{D}^{\alpha} \models Poss(\alpha, s)$ iff $\mathcal{D} \models Poss([a_1, \ldots, a_n], s)$[9], and*
2. *$\mathcal{D}^{\alpha} \models F(\vec{x}, do(\alpha, s))$ iff $\mathcal{D} \models F(\vec{x}, do([a_1, \ldots, a_n], s))$.*

When a sequence of actions is to be executed, each single action must be possible in that sequence given that the previous action was also possible. This is exactly what the formula $Poss(\alpha, s)$ in Eq. 4.2 above describes. It means in order to simulate a sequence of actions by one single new action, each precondition axiom of the original action sequence must be satisfied. Similarly, the successor state axioms must describe the outcome of the whole action sequence. Φ^{σ}_{F} is a formula which describes the successor state axiom formula for sequences of actions. In the new successor state axiom we distinguish between the new action and the old ones. In the former case we make use of the new effect formula for the action sequence with Eq. 4.3, otherwise we apply the 'old' successor state axiom.

Proof of Lemma 1. Assume for now that the formulas Π_{α} and Φ^{σ}_{F} are correct and describe the precondition as well as the sucessor state of a sequence of actions.

1. (a) We start with the proposition $\mathcal{D}^{\alpha} \models Poss(\alpha, s)$ if $\mathcal{D} \models Poss([a_1, \ldots, a_n], s)$. Equation 4.2 states that $Poss(\alpha, s) \equiv \mathcal{R}[Poss(a_1, s) \wedge \bigwedge_{i=2}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}], s)$ by construction of $\mathcal{D}^{\alpha}$. If $\mathcal{D} \models Poss([a_1, \ldots, a_n], s) \equiv Poss(a_1, s) \wedge \bigwedge_{i=2}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}], s)$ it follows from the fact that $\mathcal{D} \subset \mathcal{D}^{\alpha}$ and from Theorem 1 that also $\mathcal{D}^{\alpha} \models Poss(\alpha, s)$.

1. (b) $\mathcal{D} \models Poss([a_1, \ldots, a_n], s)$ if $\mathcal{D}^{\alpha} \models Poss(\alpha, s)$. Let M be a model and ν be a variable assignment for s such that $M, \nu \models \mathcal{D}$, and let M' be a model identical to M except for $M' \models \Pi_{\alpha}$. M and M' assign the same truth value to all formulas except for Π_{α}. It follows that $M', \nu \models \mathcal{D}^{\alpha}$. This follows directly from the proposition, Theorem 1, the UNA for actions, and the fact that $\mathcal{D}^{\alpha} = \mathcal{D} \cup \mathcal{D}^{\alpha}_{ap} \cup \mathcal{D}^{\alpha}_{ssa}$. Now suppose that $M', \nu \not\models Poss([a_1, \ldots, a_i], s)$ for some i. Then it must follow that $M', \nu \not\models Poss(\alpha, s)$ which contradicts the antecedent of the proposition. Thus, the proposition $\mathcal{D} \models Poss([a_1, \ldots, a_n], s)$ if $\mathcal{D}^{\alpha} \models Poss(\alpha, s)$ follows.

Analogously, we show the second proposition from Lemma 1.

[9] $Poss([a_1, \ldots, a_n], s)$ is an abbreviation for $Poss(a_1, s) \wedge \bigwedge_{i=2}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}], s))$.

2. (a) $\mathcal{D}^\alpha \models F(\vec{x}, do(\alpha, s))$ if $\mathcal{D} \models F(\vec{x}, do([a_1, \ldots, a_n], s))$. Similar to the precondition axiom in case 1 we argue that assumed that Eq. 4.3 is correct, $\mathcal{D}^\alpha \models F(\vec{x}, do(\alpha, s))$ as $\mathcal{D} \subset \mathcal{D}^\alpha$. If $\mathcal{D}$ entails $F(\vec{x}, do([a_1, \ldots, a_n], s))$ then, by construction of $\mathcal{D}^\alpha$ and Theorem 1, $F(\vec{x}, do(\alpha, s))$ is entailed by $\mathcal{D}^\alpha$.

2. (b) $\mathcal{D} \models F(\vec{x}, do([a_1, \ldots, a_n], s))$ if $\mathcal{D}^\alpha \models F(\vec{x}, do(\alpha, s))$. Again, let M be a model and ν be a variable assignment for s such that $M, \nu \models \mathcal{D}$, let M' be like M except for $M', \nu \models \Phi_F^\sigma$. It follows that also $M', \nu \models \mathcal{D}^\alpha$. Suppose that $M', \nu \not\models F(\vec{x}, do([a_1, \ldots, a_i], s)$ for some i. Assuming that Eq. 4.3 describes the successor state axiom for an action sequence correctly, then it also must follow that $M', \nu \not\models F(\vec{x}, do(\alpha, s))$. This leads again to a contradiction from which the proposition follows. $\square$

What remains to be shown is that Eq. 4.3 describes the successor state axiom for a sequence of actions in a correct way.

Lemma 2 *Let $\sigma = a_1; \ldots; a_n$ be sequence of primitive actions and let $\Phi_F^\sigma(s)$ be as given in Eq. 4.3 in Lemma 1. Then, $F(\vec{x}, do([a_1, \ldots, a_n], s)) \equiv \Phi_F^\sigma(s)$ describes the successor state axiom for σ.*

Proof. We show the proposition by induction over the action length. $l = 0$: It follows from Eq. 4.3 that $F(\vec{x}, s) \equiv F(\vec{x}, s)$ which is obviously true. $l = 1$: From Eq. 4.3 it follows that

$$F(\vec{x}, do(a_1, s)) \equiv F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a_1, s) \vee \gamma_F^+(\vec{x}, a_1, s)$$

which is the normal form for successor state axioms. $l = n + 1$: Assume that Eq. 4.3 describes the successor state axiom for F mentioning the action sequence $a_1; \ldots; a_n$. According to Eq. 4.3

$$\begin{aligned}
&\Phi_F^\sigma(\vec{x}, do([a_1, \ldots, a_n], s)) \equiv \\
&\quad \mathcal{R}[F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a_1, s) \wedge \cdots \wedge \neg\gamma_F^-(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s)) \vee \\
&\qquad \gamma_F^+(\vec{x}, a_1, s) \wedge \neg\gamma_F^-(\vec{x}, a_2, do(a_1, s)) \wedge \neg\gamma_F^-(\vec{x}, a_3, do([a_1, a_2], s)) \wedge \cdots \vee \\
&\qquad\quad \gamma_F^+(\vec{x}, a_{n-1}, do([a_1, \ldots, a_{n-2}], s)) \wedge \neg\gamma_F^-(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s)) \vee \\
&\qquad \gamma_F^+(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s))].
\end{aligned} \tag{4.4}$$

Now suppose the action a_{n+1} is executed after the sequence $a_1; \ldots; a_n$. According to the normal form for successor state axioms the fluent F has the value

$$\begin{aligned}
&F(\vec{x}, do([a_1, \ldots, a_n, a_{n+1}], s)) \equiv \\
&\quad F(\vec{x}, do([a_1, \ldots, a_n], s)) \wedge \neg\gamma_F^-(\vec{x}, a_{n+1}, do([a_1, \ldots, a_n], s)) \vee \gamma_F^+(\vec{x}, a_{n+1}, do([a_1, \ldots, a_n], s))
\end{aligned}$$

Substituting the right-hand side of the equivalence in Eq. 4.4 for $F(\vec{x}, do([a_1, \ldots, a_n], s))$ yields

$$\begin{aligned}
&F(\vec{x}, do([a_1, \ldots, a_n, a_{n+1}], s)) \equiv \\
&\qquad [\ \mathcal{R}[F(\vec{x}, s)) \wedge \neg\gamma_F^-(\vec{x}, a_1, s) \wedge \cdots \wedge \neg\gamma_F^-(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s)) \vee \\
&\qquad\quad \gamma_F^+(\vec{x}, a_1, s) \wedge \neg\gamma_F^-(\vec{x}, a_2, do(a_1, s)) \wedge \cdots \vee \\
&\qquad\qquad \gamma_F^+(\vec{x}, a_{n-1}, do([a_1, \ldots, a_{n-2}, s])) \wedge \neg\gamma_F^-(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s)) \vee \\
&\qquad\qquad\quad \gamma_F^+(\vec{x}, a_n, do([a_1, \ldots, a_{n-1}], s))]] \wedge \\
&\qquad \neg\gamma_F^-(\vec{x}, a_{n+1}, do([a_1, \ldots, a_n], s)) \vee \gamma_F^+(\vec{x}, a_{n+1}, do([a_1, \ldots, a_n], s))
\end{aligned} \tag{$*$}$$

Distributing the formula $\neg\gamma_F^-(\vec{x}, a_{n+1}, do([a_1, \ldots, a_n], s))$ $(*)$ over the disjunction yields Eq. 4.3 for the action sequence $a_1, \ldots, a_{n+1}$. Thus, by induction it is shown that Eq. 4.3 describes the effects of an action sequence w.r.t. F's value.

$\square$

Corollary 1 *Let $\sigma = a_1; \ldots; a_n$ be a sequence of primitive actions, and let α be a new action with precondition axioms and effect axioms as in Lemma 1. Then $\mathcal{D}^\alpha \models \exists s''.Do(\alpha, s, s'')$ iff $\mathcal{D} \models \exists s'.Do(a_1; \ldots; a_n, s, s')$.*

As a direct consequence of Lemma 1 and Lemma 2 it follows that sequences of actions can be modeled by a primitive action having the same effects and equivalent preconditions. Next, we consider the case of how a test action can be simulated by a single primitive action.

Lemma 3 *Let $\sigma = \varphi?$ be a test action. Let α be new primitive action with a precondition axiom $Poss(\alpha, s) \equiv \varphi$ and no effects, i.e. $F(\vec{x}, do(\alpha, s)) \equiv F(\vec{x}, s)$. Let $\mathcal{D}^\alpha = \mathcal{D} \cup \{Poss(\alpha, s)\}$. Then $\mathcal{D}^\alpha \models \exists s''.Do(\alpha, s, s'')$ iff $\mathcal{D} \models \exists s'.Do(\varphi?, s, s')$.*

Proof. The semantic definition of a test action yields that a test succeeds if $\mathcal{D} \models \exists s.Do(\varphi?, s, s) \equiv \varphi[s]$. From the construction of $\mathcal{D}^\alpha$ it follows that $\mathcal{D}^\alpha \models (Poss(\alpha, s) \equiv \varphi[s])$ iff $\mathcal{D} \models \varphi[s]$. As the action α has no effects, it follows that for all fluents F and all actions a: $\mathcal{D} \models F(\vec{x}, do(a, s))$ iff $\mathcal{D}^\alpha \models F(\vec{x}, do(\alpha, s))$. $\square$

Lemma 4 *Let $\sigma = (\textbf{if } \varphi \textbf{ then } a_1 \textbf{ else } a_2 \textbf{ endif})$ be a conditional, a_1, a_2 be primitive actions, $\Phi_{a_1}(s) \equiv \mathcal{R}[F(\vec{x}, do(a_1, s))]$ and $\Phi_{a_2}(s) \equiv \mathcal{R}[F(\vec{x}, do(a_2, s))]$. Further, let α be a new primitive action with $Poss(\alpha, s) \equiv \mathcal{R}[\varphi[s] \wedge Poss(a_1, s) \vee \neg\varphi[s] \wedge Poss(a_2, s)]$ its precondition axiom and $\mathcal{D}_{ssa}^\alpha$ a set of successor state axioms*

$$\begin{aligned} F(\vec{x}, do(a, s)) \quad &\equiv \quad (a \neq \alpha) \wedge [F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \vee \gamma_F^+(\vec{x}, a, s)] \vee \\ &\qquad (a = \alpha) \wedge (\varphi[s] \wedge \Phi_{a_1}(s) \vee \neg\varphi[s] \wedge \Phi_{a_2}(s)). \end{aligned}$$

Let $\mathcal{D}^\alpha = \mathcal{D} \cup \{Poss(\alpha, s)\} \cup \mathcal{D}_{ssa}^\alpha$. Then, $\mathcal{D}^\alpha \models \exists s''.Do(\alpha, s, s'')$ iff $\mathcal{D} \models \exists s'.Do(\textbf{if } \varphi \textbf{ then } a_1$ $\textbf{else } a_2 \textbf{ endif}, s, s')$.

Proof. For the conditional, we can derive

$$\begin{aligned} &\mathcal{D} \models \exists s'.Do(\textbf{if } \varphi \textbf{ then } a_1 \textbf{ else } a_2 \textbf{ endif}, s, s') \\ &\quad \overset{def}{=} \quad Do([\varphi?; a_1] | [\neg\varphi?; a_2], s, s') \equiv Do(\varphi; a_1, s, s') \vee Do(\neg\varphi; a_2, s, s') \\ &\quad \equiv \quad \exists s''.Do(\varphi?, s, s'') \wedge Do(a_1, s'', s') \vee \exists s''.Do(\neg\varphi?, s, s'') \wedge Do(a_s, s'', s') \\ &\quad \equiv \quad \exists s''.\varphi[s] \wedge s'' = s \wedge Poss(a_1, s'') \wedge s' = do(a, s) \vee \\ &\qquad\qquad \exists s''.\neg\varphi[s] \wedge s'' = s \wedge Poss(a_2, s'') \wedge s' = do(a_2, s) \end{aligned}$$

This means that either $\mathcal{D} \models \exists s.\varphi[s] \wedge Poss(a_1, s)$ or $\mathcal{D} \models \exists s.\neg\varphi[s] \wedge Poss(a_2, s)$. Regarding the logical formula derived from executing the action α

$$\begin{aligned} &\mathcal{D}^\alpha \models \exists s''.Do(\alpha, s, s'') \\ &\quad \overset{def}{=} \quad \exists s''.Poss(\alpha, s) \wedge s'' = do(\alpha, s) \\ &\quad \equiv \quad \mathcal{R}[\varphi[s] \wedge Poss(a_1, s) \vee \neg\varphi[s] \wedge Poss(a_2, s)] \wedge s'' = do(\alpha, s) \end{aligned}$$

it is apparent that either $\varphi[s] \wedge Poss(a_1, s)$ or $\neg\varphi[s] \wedge Poss(a_2, s)$ is entailed by $\mathcal{D}^\alpha$. As $\mathcal{D}^\alpha$ is constructed from $\mathcal{D}$ it follows that $\mathcal{D} \models \exists s.\varphi[s] \wedge Poss(a_1, s) \vee \neg\varphi[s] \wedge Poss(a_2, s)$ iff $\mathcal{D}^\alpha \models \exists s.Poss(\alpha, s)$.

The effects of the conditional are either the effects of action a_1 or of action a_2 depending on φ, i.e. $\mathcal{D} \models \exists s.\varphi[s] \wedge [F(\vec{x}, do(a_1, s)) \equiv \Phi_{a_1}(s)]$ or $\mathcal{D} \models \exists s.\neg\varphi[s] \wedge [F(\vec{x}, do(a_2, s)) \equiv \Phi_{a_2}(s)]$. For the new action it holds that $\mathcal{D}^\alpha \models F(\vec{x}, do(\alpha, s)) \equiv (\varphi[s] \wedge \Phi_{a_1}(s) \vee \neg\varphi[s] \wedge \Phi_{a_2})$. Thus, $\mathcal{D}$ and $\mathcal{D}^\alpha$ entail the same effects for the conditional and for the new primitive action, resp. The other direction is trivial as if $\mathcal{D}^\alpha$ exists and entails the effects for action α then there must also exists $\mathcal{D}$ entailing either the effects of action a_1 or a_2 depending on φ. Hence, the proposition follows. $\square$

We have shown that sequences of primitive actions, conditionals mentioning primitive and test actions can be modeled by a new primitive action with equivalent effects and preconditions. Next, we show that we can model restricted programs without $sprob$'s by a single primitive action.

Theorem 2 *For every restricted program σ without $sprob$ there is an augmented basic action theory $\mathcal{D}^\sigma$ and a primitive action α such that for all fluents F*

$$\mathcal{D}^\sigma \models \forall \vec{x}, s'.Do(\sigma, s, s') \quad \supset \quad [Poss(\alpha, s) \wedge F(\vec{x}, s') \equiv F(\vec{x}, do(\alpha, s))]$$

Proof. We show the proposition by induction over the program structure of σ.

1. Let $\sigma = a$ be a single primitive action. The proposition follows from Corollary 1 with α being constructed as in Lemma 1. If s' exists then a is executable and $s' = do(a, s)$ (cf. also Eq. 3.15 on page 45). Let α be a new action constructed from a as in Lemma 1 and sequence length 1. From the construction of α it follows that (disregarding unique names) $F(\vec{x}, s') \supset F(\vec{x}, do(\alpha, s))$ for each F in $\mathcal{D}^\alpha$. The antecedent of the implication in the proposition assures that the situation s' is executable, i.e. $Poss(a, s)$ holds; therefore also $Poss(\alpha, s)$ holds by the construction given in Lemma 1. The other direction directly follows from Lemma 1 and Corollary 1.

2. Let $\sigma = \varphi?$. Let α be a primitive action constructed as in Lemma 3. In this case $s' = s$ iff $\varphi[s]$ holds. Thus, $F(\vec{x}, s') = F(\vec{x}, s)$. From the construction of the action α according to Lemma 3 we get that $Poss(\alpha, s) \equiv \varphi[s]$, further, action α does not change F's successor state axiom. This means that if φ holds, the right-hand side of both equivalences hold, and with this the proposition follows.

3. Let $\sigma = \textbf{if } \varphi \textbf{ then } a_1 \textbf{ else } a_2 \textbf{ endif}$ with a_1 and a_2 being primitive actions. We construct a primitive action α as stated in Lemma 4. Either $s' = do(a_1, s)$ or $s' = do(a_2, s)$. From the existence of s' we know that either a_1 or a_2 is executable depending on the truth value of $\varphi[s]$. This means that either $\mathcal{D}^\alpha \models \exists \vec{x}, s'.\varphi[s] \wedge s' = do(a_1, s)$ or $\mathcal{D}^\alpha \models \exists \vec{x}, s'.\neg\varphi[s] \wedge s' = do(a_2, s)$. From this follows that either $F(\vec{x}, do(a_1, s))$ is true (or false), or $F(\vec{x}, do(a_2, s))$ is true (or false), resp. If $\mathcal{D}^\alpha \models \varphi[s]$ then $F(\vec{x}, do(\alpha, s)) \equiv F(\vec{x}, do(a_1, s))$, otherwise $F(\vec{x}, do(\alpha, s)) \equiv F(\vec{x}, do(a_2, s))$. Finally, from the construction of α given in Lemma 4 it follows that $Poss(\alpha, s) \equiv \varphi[s] \wedge Poss(a_1, s) \vee \neg\varphi[s] \wedge Poss(a_2, s)$. Thus, if $\mathcal{D}^\alpha \models \forall \vec{x}, s.\varphi[s] \wedge s' = do(a_1, s) \supset F(\vec{x}, s') \equiv F(\vec{x}, do(\alpha, s))$ then also $\mathcal{D}^\alpha \models \forall \vec{x}, s.\varphi[s] \wedge s' = do(a_1, s) \supset F(\vec{x}, s') \equiv Poss(\alpha, s) \wedge F(\vec{x}, do(\alpha, s))$.

In the induction step we have to distinguish the same cases. We begin with the case of a primitive action.

1. Now let $\sigma = a; \varrho$. Assume that the induction hypothesis holds, that is, an augmented basic action theory $\mathcal{D}^\alpha$ and a new primitive action α for the program ϱ exists. According to Lemma 1 there is an augmented action theory $\mathcal{D}^\sigma$ which entails a new action β having the same preconditions and effects as the action sequence. Hence, the proposition follows.

2. Let $\sigma = \varphi?; \varrho$. Assuming that the induction hypothesis holds we have a primitive action α and basic action theory $\mathcal{D}^\alpha$ which models the preconditions and effects of ϱ. With Lemma 3 we yield a new primitive action β with $\mathcal{D}^\beta \models \exists s. Do(\varphi?, s, s')$. According to the previous case for primitive actions, we now can construct another action γ and a basic action theory $\mathcal{D}^\gamma$ with: $\mathcal{D}^\gamma \models \forall \vec{x}, s'. Do(\sigma, s, s') \supset [Poss(\gamma, s) \land F(\vec{x}, s') \equiv F(\vec{x}, do(\gamma, s))]$.

3. Let $\sigma = \mathbf{if}\ \varphi\ \mathbf{then}\ \varrho_1\ \mathbf{else}\ \varrho_2\ \mathbf{endif}$. Assume that the induction hypothesis holds, i.e. that we have two primitive actions α_1 and α_2 together with their basic action theories $\mathcal{D}^{\alpha_1}$ and $\mathcal{D}^{\alpha_2}$. α_1 is a new primitive action for ϱ_1, α_2 for ϱ_2. We now construct, according to Lemma 4, a new primitive action β. As has been shown in Lemma 4, constructing a primitive action from a conditional consisting of primitive actions is correct. Therefore, $\mathcal{D}^\beta \models \forall \vec{x}, s'. Do(\sigma, s, s') \supset [Poss(\beta, s) \land F(\vec{x}, s') \equiv F(\vec{x}, do(\beta, s))]$.

□

The above theorem shows that each restricted program without $sprob$ can be simulated by a single primitive action. Now, we have the prerequisites to see that the $choice$, $prob$, and $senseCond$ predicates as used in Reiter's approach to stochastic action models can be equivalently expressed with our new $sprob$ statement.

Theorem 3 *Let $\mathcal{D}$ be a basic action theory, let $n_1, \ldots, n_k$ be primitive actions, $p_1, \ldots, p_k$ be probabilities with $p_i \in [0,1]$ and $\sum_i p_i = 1$, and $\varphi_1(s), \ldots, \varphi_k(s)$ situation-dependent logical formulas. Further, let $choice(A, a, s) \overset{def}{=} \varphi_1(s) \supset (a = n_1) \land \cdots \land \varphi_k(s) \supset (a = n_k)$ and $prob(n_1, a, s) = p_1, \ldots, prob(n_k, a, s) = p_k$, and* **procmodel** *$A$ $sprob(\{(n_1, p_1, \varphi_1), \ldots, (n_k, p_k, \varphi_k)\})$* **endprocmodel** *be a stochastic procedure model. Then,*

$$
\begin{aligned}
&\mathcal{D} \models \exists(\pi', \pi'', v', v'', pr', pr'').\\
&\quad BestDoAux(sprob(\{(n_1, p_1, \varphi_i), \ldots, (n_k, p_k, \varphi_k)\}), a, p, s, h, \pi', v', pr') \land\\
&\qquad BestDoAux(\{n_1, \ldots, n_k\}, a, p, s, h, \pi'', v'', pr'') \supset\\
&\qquad\quad \pi' = \pi'' \land v' = v'' \land pr' = pr''
\end{aligned}
$$

Proof. In the following we show that the policies, values, and success probabilities are the same, regardless if we calculate the policy with the original approach or with stochastic procedures, given the premises above. This result then shows the correctness of our stochastic procedure model when applied the same way as Reiter's stochastic action approach. (1) Assume that the respective outcome action is not possible. Then,

$$
\begin{aligned}
&BestDoAux(\{n_k\}, a, p, s, h, \pi, v, pr) \overset{def}{=}\\
&\quad \neg Poss(n_k, s) \land \pi = Stop \land v = 0 \land pr = 0
\end{aligned}
\tag{4.5}
$$

for the original stochastic action, and

$$\begin{aligned}
&\mathit{BestDoAux}(\mathit{sprob}(\{(n_k, p_k, \varphi_k)\}), a, p, s, h, \pi, v, pr) \overset{\mathit{def}}{=} \\
&\quad \neg(\exists\delta, s'.\mathit{trans}^*(n_k, s, \delta, s') > 0 \wedge \mathit{Final}(\delta, s')) \wedge \pi = \mathit{Stop} \wedge v = 0 \wedge pr = 0 \quad (4.6)
\end{aligned}$$

It was shown by De Giacomo et al. (2000) and Grosskreutz (2002) that $\mathcal{D} \models \mathit{Do}(\varrho, s, s') \equiv \mathit{trans}(\varrho, \delta, s', \delta') > 0 \wedge \mathit{Final}(\varrho, s')$. As, $\mathcal{D} \models \forall s.\neg\mathit{Poss}(n_k, s) \equiv \neg(\exists\delta.\mathit{trans}(n_k, s, \delta, do(n_k, s)) > 0 \wedge \mathit{Final}(\delta, s'))$, the same policies, values, and success probabilities are generated in the case that the single primitive action n_k is not possible. Thus, both right-hand sides of Eq. 4.5 and 4.6 are equivalent. Now, assume n_k is possible. Then, with the original approach, the formula

$$\begin{aligned}
&\mathit{BestDoAux}(\{n_k\}, a, p, s, h, \pi, v, pr) \overset{\mathit{def}}{=} \\
&\quad \mathit{Poss}(n_k, s) \wedge \mathit{senseCond}(n_k, \varphi_k) \wedge \exists\pi', v', pr'.\mathit{BestDo}(p, do(n_k, s), h, \pi', v', pr') \wedge \\
&\qquad \pi = \varphi_k?; \pi' \wedge v = v' \cdot \mathit{prob}(n_k, a, s) \wedge pr = pr' \cdot \mathit{prob}(n_k, a, s)
\end{aligned}$$

is expanded, while our approach expands

$$\begin{aligned}
&\mathit{BestDoAux}(\mathit{sprob}(\{(n_k, p_k, \varphi_k)\}), p, s, h, \pi, v, pr) \overset{\mathit{def}}{=} \\
&\quad \exists\delta, s'.\mathit{trans}^*(n_1, s, \delta, s') > 0 \wedge \mathit{Final}(\delta, s') \wedge \\
&\qquad \exists\pi', v', pr'.\mathit{BestDo}(p, s', h, \pi', v', pr') \wedge \pi = \varphi_k?; \pi' \wedge v = v' \cdot p_k \wedge pr = pr' \cdot p_k
\end{aligned}$$

As s' in the latter case is $s' = do(n_k, s)$ and $\mathit{prob}(n_k, s, a) = p_k$, it is obvious that both formulas calculate the same policies, values, and success probabilities.

Now assume that there is more than one possible outcome action given, and assume that the induction hypothesis holds, i.e. both models generate the same policies, values, and probabilities up to the current action. Now, we take a next primitive outcome action n_{k+1} into account. Again, we first review both BestDo formulas. The original approach yields:

$$\begin{aligned}
&\mathit{BestDoAux}(\{n_k, n_{k+1}\}, a, p, s, h, \pi, v, pr) \overset{\mathit{def}}{=} \\
&\quad \mathit{Poss}(n_k, s) \wedge (\exists\pi', v', pr').\mathit{BestDoAux}(\{n_k, n_{k+1}\}, p, s, h, \pi', v', pr') \wedge \\
&\quad \exists\pi_k, v_k, pr_k.\mathit{BestDo}(p, do(n_k, s), h-1, \pi_k, v_k, pr_k) \wedge \mathit{senseCond}(n_k, \varphi_k) \\
&\quad \pi = \textbf{if } \varphi_k \textbf{ then } \pi_k \textbf{ else } \pi' \textbf{ endif} \wedge \\
&\quad v = v' + v_k \cdot \mathit{prob}(n_k, a, s) \wedge pr = pr' + p_k \cdot \mathit{prob}(n_k, a, s).
\end{aligned}$$

With the novel approach one gets the formula

$$\begin{aligned}
&\mathit{BestDoAux}(\mathit{sprob}(\{(n_k, p_k, \varphi_k), (n_{k+1,}p_{k+1}, \varphi_{k+1})\}, p, s, h, \pi, v, pr) \overset{\mathit{def}}{=} \\
&\quad \exists\delta, s'.\mathit{trans}^*(n_k, s, \delta, s') > 0 \wedge \mathit{Final}(\delta, s') \wedge \\
&\qquad \exists\pi'', v'', pr''.\mathit{BestDoAux}(\{(n_{k+1}, p_{k+1}, \varphi_{k+1})\}, p, s, h, \pi'', v'', pr'') \wedge \\
&\qquad \exists\pi', v', pr'.\mathit{BestDo}(p, s', h-1, \pi', v', pr') \wedge \\
&\qquad \pi = \textbf{if } \varphi_k \textbf{ then } \pi' \textbf{ else } \pi'' \textbf{ endif} \wedge \\
&\qquad v = v'' + p_k \cdot v' \wedge pr = pr'' + p_k \cdot pr'
\end{aligned}$$

As in the latter case $s' = do(n_k, s)$ both formulas are equivalent and yield the same policy, value, and probability of success. In the next recursion the already shown case of a single primitive action occurs showing that both formulas are equivalent. The case that n_k is not possible is similar to the case of a single non-possible outcome action and is therefore omitted here. Hence, for primitive actions, our stochastic procedure approach yields the same results as the original one. □

From the previous results it is obvious that we can use our stochastic procedure model basically the same way as the original stochastic action model. As a real extension to the original model note that our definition of restricted programs also allows *sprob* statements to appear inside conditionals. With this we have the possibility to formulate conditional stochastic models with several sets of outcomes. Consider the following model:

```
procmodel $A$
    if $\psi$ then $sprob((n_1, p_1, \varphi_1), \ldots, (n_k, p_k, \varphi_k))$
    else $sprob((n_{k+1}, p_{k+1}, \varphi_{k+1}), \ldots, (n_{k+m}, p_{k+m}, \varphi_{k+m}))$
endprocmodel
```

Depending on ψ either the outcomes $n_1, \ldots, n_k$ are chosen, or the outcomes $n_{k+1}, \ldots, n_{k+m}$ for the case that $\neg\psi$ holds. Thus, at a time always one outcome set is 'active'. With this, prerequisites like $\sum_i p_i = 1$ are satisfied when executing the policy. To formally satisfy these properties, one has to transform such action models in the following way: (1) For each different set of outcomes introduce a new stochastic action A_i containing only one *sprob* statement each; (2) rewrite the original procedure model such that it contains the conditional in the form

```
procmodel $A$
    if $\psi$ then $A_1$ else $A_2$ endif
endprocmodel
```

This yields a number of stochastic actions with only one *sprob* statement for which the *choice* and *prob* statements can be easily established. In the interpretation of the stochastic procedure the last mentioned transformation is opaque as *procmodel*'s have a similar semantics as procedures in GOLOG. The name of the procedure is expanded to its body and the body is further interpreted. Therefore, the conditional in the procedure model of A is interpreted as if it would appear in the DTGOLOG code.

4.3 Speeding Up Planning

So far we showed the semantics of the READYLOG constructs and the integration of decision-theoretic planning à la DTGOLOG in the READYLOG framework. We are interested in designing a robot programming language which supports the development of controller programs for robots acting in dynamic real-time domains. The aspect of the real-time ability is very important for fast robots in dynamic domains. Therefore, we must think about how computation times for policy generation could be further reduced in practice.

In the following we present several extensions which speed up the calculation of an optimal policy. The first extension, the macro actions or options, is an approach to reduce the action space for solving MDPs. The second extension, caching, reduces the computation time of the

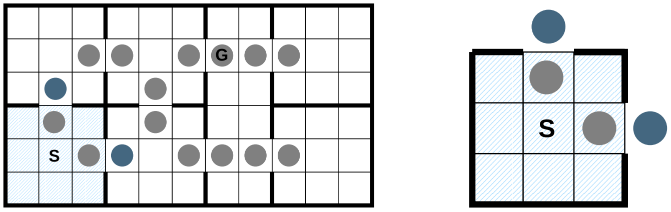


Figure 4.6: The Maze Domain with Macro-Actions (after (Hauskrecht et al. 1998)).

forward search planning algorithm by exploiting the fact that states of the MDP to be solved are represented by multiple different action histories. Further, we present some ideas to prune the tree while calculating a policy. While optimality cannot be guaranteed any longer it saves a lot of computation time. Finally, we extend the forward search value iteration algorithm to an any-time algorithm. With this extension it is possible to set a time bound instead of a horizon. A policy is then calculated until the bound for the computation is reached.

4.3.1 Options

When solving an MDP in READYLOG one uses nondeterministic choices of actions for expressing the choice points of the agents where the different choices are optimized away, and stochastic action outcomes to describe the uncertainty of the agent about the success of its actions. For the forward search value iteration algorithm this means that it must branch at every agent choice point and at every point where nature can choose the current outcome of an action. Thus, the computation tree grows exponentially in the number of choices. To reduce the complexity the concept of macro actions from classical AI planning is adapted to the MDP context. In classical planning, a macro is defined by a sequence of actions. This sequence builds a more complex action which can be used for planning. By this hierarchy of actions, the complexity of the planning task can be reduced.

For decision-theoretic planning with stochastic action outcomes the idea of building macros as in classical planning does not apply. Instead, one has to represent macros over MDPs or options as conditional sub-plans. To illustrate the idea of options we start with a simple example.

An Example

In our grid world example we have seen that in the full planning approach the agent can choose between four actions which have four possible outcomes each. Thus, at each stage of the algorithm 16 nodes have to be expanded. Solving the navigation task for large domains becomes infeasible, even with the forward search value iteration approach from DTGOLOG. Therefore one has to identify appropriate sub-tasks to reduce the complexity of the task.

Sub-tasks for finding the way to the goal are to leave certain rooms and enter other ones. Accordingly, to reach the goal from the start state “S” one possibility is to execute the action

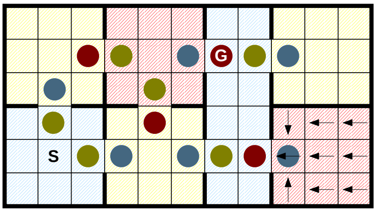


Figure 4.7: Options for the Maze66 domain from (Hauskrecht et al. 1998).

sequence

$$leaveroom\_1\_north; leaveroom\_2\_east; leaveroom\_4\_east.$$

Figure 4.6 depicts the situation. There are two possibilities to leave room 1: leave it through the northern entrance or through the eastern one. The blue spots in Figure 4.6 depict the two possibilities. The other entrance fields are marked with the gray circles. This example shows that only three actions are needed to reach the goal, instead of minimal 8 actions when using the basic actions from the set A_{base} (defined on page 72 where we introduced the maze domain). We can obviously reduce the number of expanded nodes in the calculation tree with these actions, when we are searching for the optimal policy for the navigation problem.

Clearly, one could define basic actions for leaving a room. Then, one additionally has to specify the behavior the agent should take when it is located inside a room. But this is not needed as we can make use of decision-theoretic planning. This, moreover, yields optimal behavior for leaving a room. We can relax the original problem to the problem of leaving Room 1, solve this problem and save the policy, the value, and the probability of success. Later, when solving the original problem we could use the results of solving the MDP which leaves room 1. Figure 4.7 shows the solution of one of the identified sub-tasks for Room 6. The arrows represent the optimal policy for leaving Room 6.

As Room 1 has two different doors to neighboring rooms we have to define two different options, one for leaving through the northern door, and one for leaving the room through the eastern door.

Why do we have to distinguish between both doors? The reason is even if we want to leave the room through the northern door, it might be the case that the agent ends up in room 3 due to failing basic actions. Consider the agent being located on position $(3, 2)$. The optimal policy should take the agent to position $(3, 3)$ with a go_up action. But if the action fails and nature chooses a go_right action we end up in room 3. This is obviously not what we wanted. Therefore, this case should be declared as a failure case for the macro action $leaveroom\_1\_north$.

For each room we have to identify one macro action for each door, solve the MDP of the sub-task and store the result. The results can then be re-used for solving the task of reaching the goal position.

Theoretical Background

The idea of using macro actions is not new, and also in the context of MDPs some work on that matter exists. For example Sutton et al. (1999) investigate macro action, so-called options, in the reinforcement learning context in a similar domain.

Definition 4 (Options (Sutton et al. 1999)) *An option O is defined as the triple $\langle I, \pi, \beta\rangle$, over an MDP $M = \langle S, A, T, R\rangle$, where*

- *$I \subseteq S$ the initiation set,*
- *$\pi : S \times A \to [0, 1]$ a policy, and*
- *$\beta : S^+ \to [0, 1]$ a termination condition.*

If the agent is in a state from the initiation set, the option is applicable. If the agent decides to execute the option, actions are selected according to the policy π until the option terminates stochastically according to β. A natural assumption is that for all states s with $\beta(s) < 1$: $s \in I$. Thus, all states s' with $s' \in S\backslash I$ have a termination probability of 1 and it suffices to define π over states in I. Sutton et al. (1999) prove that an option equipped with an appropriate transition model and a reward function can be used together with primitive actions to generate an optimal policy for the original problem.

The work of (Hauskrecht et al. 1998) also bases on Sutton et al. (1999). They define a macro-action as a policy for a certain region of the state space. The *region-based decomposition* of an MDP is a partitioning $\Pi = \{S_1, \ldots, S_n\}$ where the S_i are called *region*. Each region has a set of *entry states* $EPer(S_i)$ and of *exit states* $XPer(S_i)$ (the marked fields in Figure 4.7). Similarly, they define the termination condition as

$$\beta(s) = \begin{cases} 0.0, & s \in I \\ 1.0, & \text{otherwise} \end{cases}$$

and the policy as

$\pi : I \times A \to \{0, 1\}$ and thus as $\pi : I \to A$.

With this definition of a policy one can define the transition model of an MDP over policies. This allows for using options the same way like primitive actions.

Definition 5 (Discounted Transition Model (Hauskrecht et al. 1998)) *A discounted transition model $Tr_i(\cdot, \pi_i, \cdot)$ for a macro action π_i defined on region S_i is a mapping $Tr_i : S_i \times XPer(s_i) \to [0, 1]$ such that*

$$\begin{aligned} Tr_i(s, \pi_i, s') &= \mathrm{E}\left[\gamma^{t-1} Pr(s^\tau = s' | s^0 = s, \pi_i)\right] \\ &= \sum_{t=1}^{\infty} \gamma^{t-1} Pr(\tau = t, s^t = s' | s^0 = s, \pi_i) \end{aligned}$$

where τ denotes the time of termination of π_i. A discounted reward model $R_i(\cdot, \pi_i)$ for π_i is a mapping $R_i : S_i \to \mathrm{I\!R}$ such that

$$R_i(s, \pi_i) = \mathrm{E}_\tau \left[\sum_{t=0}^{\tau} R(s^\tau, \pi_i(s^t) | s^0 = s, \pi)\right]$$

For each state $s \in S_i$ the probability of leaving this region through an exit state $s' \in XPer(S_i)$ is defined by this model. Note that the model is discounted by the expected time until leaving the region. Also note that the transition model is defined over a policy not over actions. Precup et al. (1998) showed in their Composition Theorem that it is possible to solve a discounted MDP optimally also if macro actions are used.

For each state $s \in S, s' \in XPer(s)$ the discounted probability for macro π_i satisfies

$$Tr_i(s, \pi_i(s), s') = Tr(s, \pi_i(s), s') + \gamma \sum_{s'' \in S_i} Tr(s, \pi_i(s), s'')\, Tr_i(s'', \pi_i, s'),$$

defining a system of $|S_i|$ linear equations for each exit state s'. Tr is the transition model for the global MDP, while Tr_i is the model for the macro π_i.

So, the idea of using macro actions in the MDP context is to define local MDPs for partitions of the state space together with their peripheral states, calculate the solution for the local MDP and use it again when solving the global MDP. Let S_i be a region of MDP $M = \langle S, A, Tr, R\rangle$ and let $\sigma : XPer(S_i) \to \mathbb{R}$ be a seed function for S_i. The local MDP $M_i(\sigma)$ associated with S_i and σ consists of (1) the state space $S_i \cup XPer(S_i) \cup \{\alpha\}$ where α is a new reward-free absorbing state, (2) actions, dynamics, and rewards associated with S_i in M, (3) a reward $\sigma(s)$ associated with each $s \in XPer(S_i)$, (4) an extra single cost-free action applicable at each $s \in XPer(S_i)$ that leads with certainty to α.

For assigning the local reward function usually one assumes that the agent always wants to leave the region S_i via an exit state. Therefore, one assigns a high reward to these states to achieve goal-directed behavior. When the option is used to solve the global MDP the policy of the macro is taken and the value for the policy is calculated based on the original value function. With the previous definitions and the definition of an abstract MDP given below, we have set all preliminaries to define options in the READYLOG context.

Definition 6 (Abstract MDP (Hauskrecht et al. 1998)) *Let $\Pi = \{S_i, \ldots, S_k\}$ be a decomposition of MDP $M = \langle S, A, Tr, R\rangle$, and let $\mathcal{A} = \{A_i \,|\, i \leq n\}$ be a collection of macro action sets, where $A_i = \{\pi_i^1, \ldots, \pi_i^{n_i}\}$ is a set of macros for region S_i. The abstract MDP $M' = \langle S', A', Tr', R'\rangle$ induced by Π and $\mathcal{A}$ is defined by:*

- $S' = Per_\Pi(S) = \bigcup_{i \leq n} EPer(S_i)$
- $A' = \bigcup_i A_i$ *with $\pi_i^k \in A_i$ feasible only at states $s \in EPer(S_i)$*
- $T'(s, \pi_i^k, s')$ *is given by the discounted transition model for π_i^k (Def. 5), for any $s \in EPer(S_i)$ and $s' \in XPer(S_i)$; $T'(s, \pi_i^k, s') = 0$ for any $s' \notin XPer(S_i)$*
- $R'(s, \pi_i^k)$ *is given by the discounted reward model for π_i^k, for any $s \in EPer(S_i)$*

Summarizing, for using options one first has to identify suitable sub-tasks and define partitions of the state space of the MDP including the peripheral states, i.e. the states where the option is applicable and the states where the option terminates. Further one has to provide a discounted transition model for the option as well as a discounted reward model. In an abstract MDP basic actions as well as macros which are defined over partitions of the state space, can be used to solve the planning problem.

Options in Readylog

Our approach to using options in READYLOG is similar to the approach of Hauskrecht et al. (1998) and incorporates ideas of Sutton et al. (1999). We also define and solve local MDPs which describe a sub-problem of the global task. The solution of the sub-problem then can be used to solve the original problem instance. Similar to 'classical' solution strategies for MDPs, the restriction of options in READYLOG is that the method relies on enumerable state spaces. This is in contrast to the forward search value iteration known from DTGOLOG which can handle infinite state spaces. The restriction is contributed to the fact that convergence (and thus optimality) can only be guaranteed for discounted MDPs.

In the following we formally define options in the READYLOG context and present a modified version of the value iteration algorithm which is based on the action theories of the situation calculus.

Definition 7 (READYLOG Option) *An option o is the triple $\langle \varphi(s), \pi, \beta \rangle$ where $\varphi(s)$ is a logical formula describing the initiation set. An option is applicable iff $\varphi(s)$ holds. π is a* READYLOG *policy, and β is a logical formula denoting the termination of an option. An option is said to be of level 1 iff its respective policy π only contains primitive or stochastic actions. An option is of level n iff its respective policy program π only contains options of level $n-1$.*

This definition yields a hierarchy over options. The policy for an option may only mention primitive actions or options of a lower level. How do we obtain the policy program π? Similar to the forward search algorithm in DTGOLOG, a READYLOG program must be specified for the option from which the optimal policy is calculated. We call this program *option skeleton*.

Definition 8 (Option Skeleton) *An option skeleton program σ of level n is a* READYLOG *program which may only mention*

- *options of level less than n,*
- *nondeterministic choices over option skeletons of level less than n*
- *conditionals* $\mathbf{if}\ \varphi\ \mathbf{then}\ \sigma_1\ \mathbf{else}\ \sigma_2\ \mathbf{endif}$, *where σ_1 and σ_2 are option skeletons of level less than n.*

An option skeleton gives the solution strategy for finding a policy for the sub-task. We have to restrict an option skeleton because we have to solve the sub-MDP induced by the option optimally as a discounted problem. In particular, no sequences of actions are allowed to conserve the Markov property.[10] In the previous section we mentioned the Composition Theorem (Precup et al. 1998) which shows that optimizing over sub-policies converges. Therefore, we have to define a discounted fully observable MDP. To be able to solve an option skeleton with ordinary value iteration techniques, we have to map situations from the situation calculus to states of the MDP, states to situations, and states to *senseEffect* conditions for conserving the full observability property:

[10] Note that we do allow macro actions of a lower level.

- In general, a state representation can be arbitrary but it is suggestive to use a factored representation (Boutilier et al. 1999) based on variables. Without loss of generality, we assume a factored representation of the state space. The factoring gives us a means to map situations into the state representation. In the situation calculus fluents describe the state of the world. Each world state can be described by the values of all fluents in a respective situation. Thus, $sitstate : S \to F_1 \times \cdots \times F_n$, where the F_i are fluent values for the situation s, describes the mapping from situation to states. Note that with defining this mapping we restrict ourselves to a finite number of fluents here.

- The value iteration algorithm works on states. To be able to describe the outcome of an option model in the situation calculus and hence use it for solving larger problems, we have to re-transform state representations into situations. As, in general, infinitely many situations describe a state, we cannot find a surjective mapping from states to situations. With assuming a factored representation of states which depends on discriminative variables, we can though restore a situation term from these. We use the special $set(f, v)$ action which sets the corresponding fluent f to value v. A state can then be described by the situation term $do(set(f_i, v_i), do(\cdots, do(set(f_1, v_1), s) \cdots)$.

- As the solution to an option (as we will see below) consists of a decision tree (conditional program) where each state of the local MDP is associated with the optimal action, we have to explicitly define the mapping between a state of the MDP and a formula how a state can be distinguished from other states.

To define an option o one has to specify the following predicates.

1. the initiation set. This is done with the predicate $option\_init(o, \varphi)$, where o is the name of the option, and φ is a logical formula which describes when the option is applicable. It is analogous to the predicate $Poss$.

2. The option skeleton $option\_skeleton(o, p, \gamma)$. o refers to the name of the option, p is an option skeleton. γ is the discounting factor.

3. The termination states $option\_beta(o, \varphi, v)$, where o names the option, φ is a logical formula which describes the terminating states, and v is a pseudo-reward. The pseudo-reward is used to create goal-directed behavior.

4. The mapping between states and situations (and vice versa) is defined by the predicate $option\_mapping(o, \sigma, p, \varphi)$, where o refers to the option name, σ describes the state of the MDP, φ is a condition which holds when the MDP is in state σ, and p is a program which transfers the current situation to the one described by σ.

5. A sense condition $option\_sense(o, a)$. Again, o refers to the option name, a is a sensing action. This sensing action is required to assure full observability of the MDP.

6. The convergence threshold $option\_epsilon(e)$. The value iteration algorithm runs as long as the difference of values are below e.

To illustrate the specification of an option we give an example from our grid world domain. We give the Prolog specification for an option which takes the agent to leave the first room through the northern door.

```
option_init(leave_room_1_n,
    and([pos = [X, Y], domain(X, [0..5]), domain(Y, [0..10]), inRoom(1)])
```

This denotes that the option leave_room_1 is applicable if the values of the position fluent pos takes values (x, y) with $0 \leq x \leq 5$ and $0 \leq y \leq 10$ (i.e. the whole maze world) and moreover the x and y take values from room 1.

```
option_beta(leave_room_1_n, inRoom(2), 100).
option_beta(leave_room_1_n, and(not(inRoom(1)), not(inRoom(2))), −100).
```

Here, we have two clauses which describe the termination condition of this option together with their pseudo-rewards. If we leave the room through the northern door (i.e. we are in room 2) the agent receives a positive rewards, if it is neither in room 1 nor in room 2 it receives a negative reward.

```
option_mapping(leave_room_1_n,[(pos, [X, Y])], [set(pos, [X, Y])], pos = [X, Y]).
```

The state variable for the maze MDP is the fluent pos. The state of the MDP can be sensed with the condition pos = [X, Y] which corresponds to φ in Item 4 above. What we further need is a program which, for a given state, creates an appropriate action history s (situation term) which assures that the condition φ holds on situation s. As described in Section 4.2.4 the easiest way to ensure that a fluent holds, is to use set actions which explicitly set a fluent to a value.

```
option_sense(leave_room_1_n, exogfUpdate).
```

The $exogfUpdate$ action is the sensing action which ensures that the position fluent in our example can be evaluated when the policy is executed on-line. option_epsilon(0.00001) sets the convergence thresholds of the value iteration to $1 \cdot 10^{-5}$.

With these predicates at hand we can solve sub-MDPs by interpreting the option skeleton. Similar to the previously described $BestDo$ predicate we define a predicate $BestDoOpt$ which recursively calculates the solution to the MDP of the option skeleton.

The value iteration algorithm given below takes a set of initiation states I, the option skeleton, a discount factor γ, the exit states β with their pseudo-rewards v, a sense action a which assures that the states can be discriminated, and a convergence threshold ε as input. The initiation set is constructed from the predicate $option\_init$. Together with the mapping from situations to states, we can generate the list of states from the condition φ. $CalculateOption(o, I, \sigma, \gamma, \beta, a, \varepsilon)$ starts the calculation of the behavior policy for σ by means of the $BestDoOpt$ predicate which we introduce below. That is, σ is interpreted by $BestDoOpt$ until convergence in the values is reached, i.e. the difference in the values for a state s is smaller than the threshold ε. $BestDoOpt$ yields the optimal policy π, the corresponding value v, and the probability of success pr together with the transition probability Π. Π keeps track of the transition probabilities between states in the underlying MDP. After the values of the behavior policy converged for all states $s \in I$, we build the run-time model

```
**Input**: Option name $o$, set of init states $I$, option skeleton $\sigma$, discount factor $\gamma$, set of exit
        states $\beta$ together with their pseudo-rewards $v$, sense action $a$, convergence threshold
        $\varepsilon$
**Output**: option model $m$
**forall** $s \in I$ **do**
    **while** $|v_{prev}(s) - v(s)| > \varepsilon$ **do**
        $(\pi, v, \Pi) := calculateOption(o, I, \sigma, \gamma, \beta, a, \varepsilon)$
        $updateStateTransitions(\pi(s), v(s), r(s), \Pi)$
        $v_{prev}(s) \leftarrow v(s)$
    **end**
**end**
$m := generateModel(o, \pi, v, pr, \Pi)$
```

of the option. In the notation of the above algorithm this is denoted the function $generateModel$ which makes use of the option name, the behavior policy as calculated with $BestDoOpt$, the optimal value, and the state transition model. In each iteration, $updateStateTransitions$ takes the optimal action for the current state denoted by $\pi(s)$, the value, the reward $r(s)$ for the current state, and the transition probability of the actual state transition as arguments. The underlying data structure stores the transition probability as well as the value and the reward for the current situation for each state and each action. The data structure is similar to a Q-table known from reinforcement learning (cf. Chapter 3.1.2). In each iteration the table is updated calling the $updateStateTransitions$ function. This table is also used to calculate the transition probabilities to the exit states β of the option. That is, we calculate the probability to reach an exit state t for each $s \in I$ with

$$T^*_\pi(s,t) = T_\pi(s,t) + \sum_{s' \in I} T_\pi(s,s') \cdot T^*_\pi(s',t), \tag{4.7}$$

where $T_\pi(s,t)$ is the one step transition model for the local MDP.[11] Solving the set of equations yields the transition probability for each state of the local MDP to an exit state.

Primitive Action

If the current action in the option skeleton is a primitive (deterministic) action, it is checked whether it is executable or not. In the former case, the algorithm goes on with interpreting the option skeleton in the successor situation, the reward for the current situation is calculated as well as the value. Note that unlike in the definition of $BestDo$, the value of the previous state is discounted by γ. In the case the action is not possible, the policy for this state is set to Nil. As mentioned before we have to maintain a set of probability values (as defined in Definition 5) given for calculating the outcome probabilities for exit states when generating the model. In the case of

[11]The one step transition model is given by the state-action table gained by the $updateStateTransition$ function.

an impossible action, the successor state s_{suc} (which is in this case s) gets the probability of 1.

$$
\begin{aligned}
&\mathit{BestDoOpt}(o,a,p,s,h,r,\Pi,d,\pi,v) \overset{def}{=}\\
&\quad s_{suc} = \mathit{sitstates}(s) \land h \geq 0\\
&\quad (\mathit{Poss}(a,s) \land \exists r', v'.\mathit{BestDoOpt}(o,a,p,do(a,s),h-1,r',\Pi,0,\pi,v') \land\\
&\quad r = \mathit{reward}(s) + r' \land v = r' + \gamma \cdot v' \lor\\
&\quad \neg \mathit{Poss}(a,s) \land r = \mathit{reward}(s) \land \pi = \mathit{Nil} \land v = 0) \land \Pi = \{(s_{suc}, 1.0)\})
\end{aligned}
$$

The predicate update is used to keep a table of states, values and probabilities. In a later step these are needed to build the model of the local MDP. Note the argument d of $\mathit{BestDoOpt}$. It is used as a program counter for remembering the current option skeleton executed. This is needed as we allow to use options hierarchically.

Stochastic Action

If the current statement is a stochastic action, it must first also be checked if the action is possible. If so, the outcomes are expanded via the $\mathit{BestDoOptAux}$ predicate, otherwise the policy is set to Nil, the value for this situation is 0. Note that differently to primitive actions the probability set Π is set to the empty set.

$$
\begin{aligned}
&\mathit{BestDoOpt}(o,a,p,s,h,r,\Pi,d,\pi,v,h) \overset{def}{=}\\
&\quad (h \geq 0 \land \mathit{Poss}(a,s) \land \exists r', \pi', v', P.\mathit{procmodel}(a,P) \land\\
&\quad\quad \mathit{BestDoOptAux}(P,p,s,h,r,\Pi,d,\pi',v) \land\\
&\quad r = \mathit{reward}(s) + r' \land v = \mathit{reward}(s) + \gamma \cdot v' \land \pi = a \lor\\
&\quad \neg \mathit{Poss}(a,s) \land r = \mathit{reward}(s) \land \pi = \mathit{Nil} \land v = 0 \land \Pi = \{\})
\end{aligned}
$$

The outcomes of an stochastic action are expanded with an auxiliary predicate. From our stochastic action model we get the list of programs describing the effect of the action, the entry probability and the sensing condition ψ which is needed to identify which outcome occurred. The predicate first expands the other possible outcomes. Then, for the first outcome in the set of outcomes for the stochastic action we distinguish the successor situation s' with trans^*, or assume the current situation if the successor situation does not exist. For this situation the state s_{suc} of the MDP is determined. In situation s' we start with the next iteration. The reward and the value for the actual outcome is calculated. The probability set Π is updated for the successor state with probability p_1. This is done by adding p_1 to the probability which is kept for s_{suc}.

$$
\begin{aligned}
&\mathit{BestDoOptAux}(o,\{(n_1,p_1,\psi_1),\ldots,(n_k,p_k,\psi_k)\},p,s,h,r,\Pi,d,\pi,v) \overset{def}{=}\\
&\quad \exists r',\Pi',\pi',v'.\mathit{BestDoOptAux}(\{(n_2,p_2,\psi_2)\ldots,(n_k,p_k,\psi_k)\},p,s,h,r',\Pi',d,\pi',v') \land\\
&\quad (\exists \varrho, s'.\mathit{trans}^*(n_1,s,\varrho,s') \lor \neg\exists \varrho, s'.\mathit{trans}^*(n_1,s,\varrho,s') \land s = s') \land\\
&\quad s_{suc} = \mathit{sitstates}(s') \land\\
&\quad \exists r^*,\Pi^*,\pi^*,v^*.\mathit{BestDoOpt}(o,p,p,s',h-1,r^*,\Pi^*,d,\pi^*,v^*) \land\\
&\quad r = r^* + p_1 \cdot r' \land v = v^* + p_1 \cdot v' \land \Pi = \Pi^* + (s_{suc}, p_1)
\end{aligned}
$$

Conditionals
Handling conditionals is the same as in DTGOLOG; if the condition holds, the if-branch p_1 is further expanded, otherwise p_2 will be followed.

$$\begin{aligned}&\mathit{BestDoOpt}(o, \mathbf{if}\ \varphi\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{endif}, s, h, r, \Pi, d, \pi, v) \overset{def}{=}\\ &\quad \varphi[s] \wedge \mathit{BestDoOpt}(o, p_1, s, h, r, \Pi, d, \pi, v) \vee\\ &\quad \neg\varphi[s] \wedge \mathit{BestDoOpt}(o, p_2, s, h, r, \Pi, d, \pi, v)\end{aligned}$$

Nondeterministic Choice of Action
Nondeterministic choices of actions are handled the same way as in DTGOLOG.

$$\begin{aligned}&\mathit{BestDoOpt}(o, (p_1|p_2), s, h, r, \Pi, d, \pi, v) \overset{def}{=}\\ &\quad (\exists v_1, \Pi_1, r_1, \pi_1.\mathit{BestDoOpt}(o, p_1, s, h, r_1, \Pi_1, d+1, \pi_1, v_1) \wedge\\ &\quad \exists v_2, \Pi_2, r_2, \pi_2.\mathit{BestDoOpt}(o, p_2, s, h, r_2, \Pi_2, d+1, \pi_2, v_2) \wedge\\ &\quad (v_1 \geq v_2 \wedge \Pi = \Pi_1 \wedge r = r_1 \wedge \pi = \pi_1 \wedge v = v_1 \vee\\ &\quad v_1 < v_2 \wedge \Pi = \Pi_2 \wedge r = r_2 \wedge \pi = \pi_2 \wedge v = v_2))\end{aligned}$$

For the projection task, we model the outcomes of an option as a stochastic action. The precondition is derived from the initiation set of the option, the outcomes are modeled inside the procedure model with an *sprob* statement. For the on-line execution of an option we must provide the optimal policy generated by the algorithm. For each option we generate a procedure as given in the procedure *option* below. The action *senseState* is a placeholder for readability reasons. It

```
proc option
  $senseState$;
  while $\varphi_0$ do
    if $\varphi_1$ then $\pi_i$ else if ... else if $\varphi_n$ endif;
    $senseState$
  endwhile
endproc
```

stands for the sensing program which was specified with $option\_sense(o, \sigma)$. After it is executed, the truth values for the sensing conditions in the current state can be determined. The condition φ_0 is the condition specified with $option\_init(o, \varphi_0)$. The φ_i are conditions which discriminate the current state the system is in. These are specified with the predicate $option\_mapping$. To illustrate the procedure we come back to the example of leaving room 1 through the northern door. The procedure which was generated by the READYLOG interpreter is given below.

Using our running example we conducted a number of experiments, the goal being at different distances from the initial position (Fig. 4.2 shows the special case of distance 8). The results are shown in Figure 4.8. The x-axis depicts the initial distance to the goal, the y-axis the running time. We compared three different approaches:

(A) calculating the optimal policy nondeterministically choosing only from the primitive actions,

```
proc(leave_room_1_n,
   [exogfUpdate,
   while(is_possible(leave_room_1_n), [
     if(pos = [0, 0], go_right, if(pos = [0, 1], go_right, if(pos = [0, 2], go_up,
       if(pos = [1, 0], go_right, if(pos = [1, 1], go_right, if(pos = [1, 2], go_right,
       if(pos = [2, 0], go_down, if(pos = [2, 1], go_right, if(pos = [2, 2], go_up
     ))))))))), exogfUpdate])]).
```

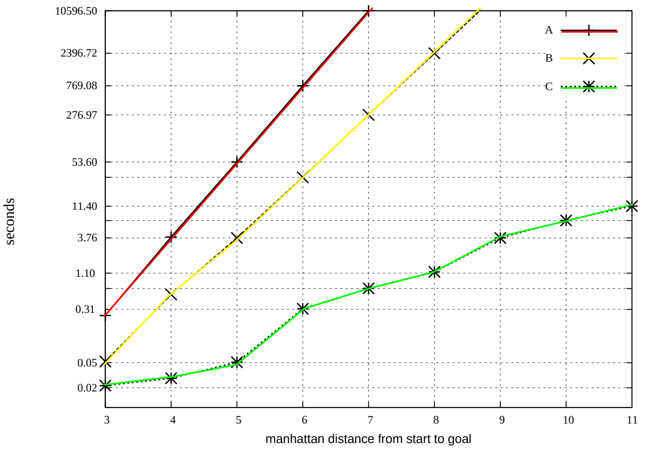


Figure 4.8: Run-time comparison between ordinary stochastic actions and options.

(B) using a set of procedures for leaving each room towards a certain neighboring room, choosing from primitive actions only in the goal room, and

(C) using options in the form of abstract stochastic actions, choosing from primitive actions only in the goal room.

Note that the y-axis of the diagram has a logarithmic scale. The speed-up from (A) to (B) shows the benefit using DTGOLOG to constrain the search space by providing fixed programs for certain sub-tasks. Interestingly, (C), that is, using abstract actions clearly outperforms (B). This is because each abstract action has only two outcomes, whereas the corresponding program provides a very fine-grained view with a huge number of outcomes that need to be considered. (We remark that while method (A) guarantees optimality, this is not necessarily so for (B) and (C). Certainly in the case of (C), this price seems worth the computational gain.)

Taking the time of calculation of all options into account (10.51 seconds in our test scenario) the use of options pays off at horizons greater than 5. Also, calculating options can be done off-line and the computation time can be neglected in case of frequent reuse.

4.3.2 Caching, Pruning and an Any-time Algorithm

Caching of Intermediate Results

To improve the interpreter's performance one has to analyze where the interpreter consumes most of its computation time. In the case of decision-theoretic solving of MDPs, the interpreter consumes most computation time in exploring the state space and calculating the values of all visited states. Calculating the value means to assign the reward function to the state, calculating the state's reward, and multiplying this reward with the probability to reach the state. The most expensive part herein from a computational point of view lies in regressing the fluents to the initial state. This is necessary to compute their current values and to complete the final reward of the current state.

Here, the idea of caching comes into play. Caching means to save and reuse the calculation of intermediate results to speed up the computation. It comes without losing expressiveness or accuracy in the numerical values and results.

Before we describe the implemented method we again point out the fine difference between the notion of state and situation. A situation is defined by the history of actions. A state is defined by the values of all fluents in it. Here, the difference becomes obvious. Different situations can describe the same state. For example, going to the right and then up describes the same state as going first up and then to the right, but the action histories are different.

The method we implemented saves the complete transition table for each state. The transition table is a way to represent the function $T(s, a, s')$. Performing action a in state s leads to state s'. We extend the function by associating the value $v_{T(s,a,s')}$ to it. Every time we have computed a value for one transition we store it along with the according transition. The states are saved as the values of each fluent of s and s'. Keep in mind that each state is completely described by all the values assigned to all fluents in the current situation. If the same transition $T(s, a, s')$ is expanded again at a later point of time the value is already known. It is read from the cache and reused without performing the same calculation again.

Why is this reasonable? The value of a state only depends on the previous action and the previous state because of the Markov assumption. Saving exactly this transition relation associated with its value is the well grounded reason which allows us to cache the transition.

The advantage of the caching idea is intuitive. In simple, discrete domains like the maze world, the execution times are much faster and the calculated values are often reused. The program gains execution time from a higher memory consumption. The resulting policy is equivalent to the one generated using the original approach.

The disadvantages are the following: instead of calculating the value of a state each time the whole state description and its associated values are saved. This results in high memory usage because each fluent has to be saved with its value. In the domain of UNREAL TOURNAMENT 2004 (see Chapter 5.1) for example, this means to save about a hundred fluents where only a small subset of them is changed from state to state. Worse is that in continuous worlds the caching method fails completely in general because only a small finite subset of the state space is visited. In this set caching succeeds only rarely.[12] In Chapter 7 we report on a qualitative abstraction

[12]This depends, of course, on the modeling of the domain but in general caching of arbitrary states fails in continuous worlds.

for the soccer domain. With this state space abstraction caching becomes available also in the continuous domain of soccer.[13]

Heuristic Pruning

Another way to reduce the computation times of the DT planning algorithm, especially when dealing with dynamic real-time domains, is that of applying heuristics and that of pruning. Pruning means to remove branches from computation, heuristics are for guiding the search through the search space by simple formulas or hints on where the solution is expected. In general, a heuristic is not always correct in its decision, although for some problems good and efficient heuristics are known. Here, we want to describe a method of pruning the decision-theoretic search tree by applying a simple heuristic. We can no longer guarantee an optimal solution. However this is reasonable due to the savings in computational complexity. The main observation we are basing this approach on is that branches with a very low probability have the same computational effort as branches with high probability. The underlying idea to save computation time is simple. Because low probable branches are occurring seldom during the real execution we do not consider them during policy generation. It seems more reasonable to generate a policy which does not handle all improbable cases in a more time efficient fashion. If in a rare occasion one of these improbable events occurs it seems more promising to generate a new policy in an efficient way than to always generate the complete and correct policy which is in general more expensive. Therefore we introduce a small bound p_{min} which represents the minimal probability of branches which is reasoned with.

In the grid world we have to find an adequate p_{min}. For example $p_{min} = p_{fail}^2$ represents the fact to forget all branches which fail two times or more often. In a complete stochastic decision tree for the grid world the fail cases of actions have huge impact on the size of the tree. Recall that associated with each action there is one successful outcome and three outcomes represent the action to fail.
In a small MDP induced by the program

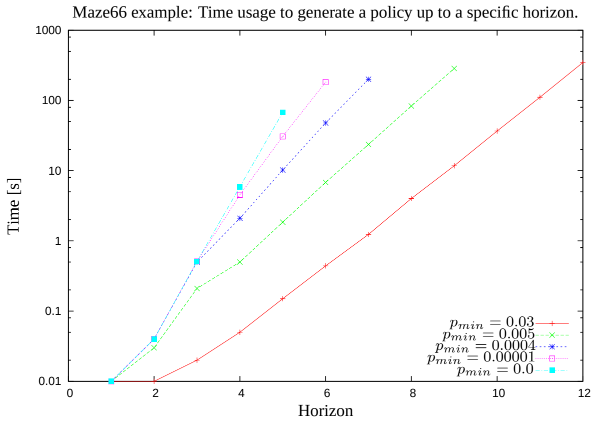
$$\mathbf{solve}(\ depth,\ \mathbf{forever\ do}\ (\text{up} \mid \text{down} \mid \text{left} \mid \text{right})\ \ \mathbf{endforever})$$

we generate all 16 outcomes at $Depth = 1$. At $Depth = 2$ the first branches are pruned and only 112 outcomes are generated.[14] In the rare case where two actions fail while executing the computed policy a re-planning step takes place and a new policy is generated.

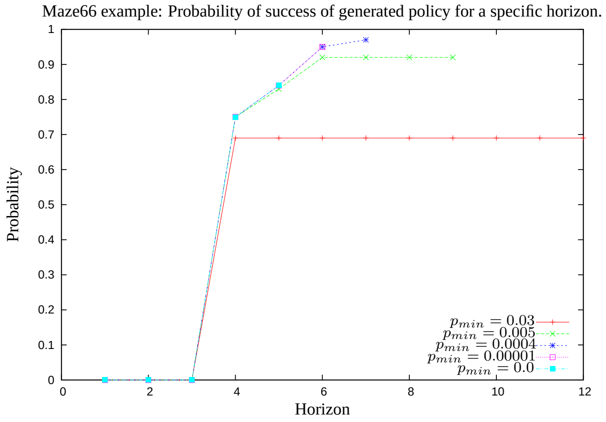
To visualize the increase in performance and the resulting consequences for the success of the computed policy see Figure 4.9(a) and Figure 4.9(b). The scenario is based on the task for the agent to move four squares in the Maze66 domain. The different lines depict different values for the minimal probability to prune with. $p_{min} = 0.0$ represents the normal policy generation without pruning. In contrast to that stands $p_{min} = 0.03$ which does not even tolerate one failing action. The generated policy assumes that no failing case occurs and therefore only those states

[13]The savings in computation time are around 1/3 in the maze domain.

[14]On level one of the tree four actions succeed and twelve fail. With each previously failing action at depth two of the tree a succeeding action is associated and not pruned. That are $12 \cdot 4$ many. With each succeeding action at depth one all outcomes of depth two are more probable than p_{min}. This are $4 \cdot 16$ many and the complete number is due to that $12 \cdot 4 + 4 \cdot 16 = 112$.



(a) Time usage for computing policies



(b) Success Probabilities resulting from pruning

Figure 4.9: This figure depicts the probabilities of the resulting policies with a given horizon. The task was to search for a policy which has to use at least four actions to be able to finish in the goal state. Lines which end are no longer computable with 1 GB of memory.

are handled which lie on the optimal path. One can see that the generation time is fast for small horizons, but the probability of success is small compared to all other test cases. $p_{min} = 0.005$ tolerates and compensates one failing action of the agent. The probabilities of succeeding in reaching the target increases as does the computation time to generate the policy. In Figure 4.9(a) you can see how much time a specific horizon takes to compute a policy for specific values of p_{min}. Note that the y-axis is scaled logarithmically. In Figure 4.9(b) one can see the success probability of the generated policy. The success probability is at the same time the value of the policy because a reward of 1.0 was given only in the target state. All other states were rewarded by 0.0 and no action costs were associated.

Nevertheless, this approach has two drawbacks. The first one was mentioned previously and is found in losing the optimality of the MDP's solution. Because we prune branches of the tree, we lose impacts from the reward function on the value function. The other drawback lies in the border p_{min}. If the horizon grows larger even actions with a relative high probability are endangered to be pruned away. This is because executing them sequentially may result in probabilities smaller than p_{min}. The designer has to take care of the relation between the maximum planning horizon and the minimal probability where to prune. Actions which succeed only with low probability are also endangered to be pruned away.

The benefits lie in the savings of computational effort. The explicit value of what is saved at computation time on the one hand depends on the modeling of the domain, and on the other hand it depends on the choice of p_{min}. In the example shown above where p_{min} corresponds to ignoring branches which contains two or more failing actions there is no gain in case the depth is one. With a depth of two only 43% of the outcomes are generated. In depth three only 640 outcomes of the original 4096 (which are only 15% of the original outcomes) have to be generated and checked. If one is able to find a suitable p_{min} for the given domain this seems to be a reasonable approach. It saves huge amounts of computation time, because it ignores improbable branches by pruning them. Nevertheless, the exponential growth in the size of the tree still remains.

Any-Time Algorithm

The need for real-time decision making in dynamic real-time domains is indispensable. Therefore we investigated possibilities to extend READYLOG to an any-time algorithm. Instead of specifying a horizon it takes a maximum run-time as argument up to which the algorithm is able to search for a solution. Afterwards the results and the best policy found so far is returned.

We adopt the idea of any-time algorithms from Boddy (1991). He describes methods concerning planning with problems which depend on time. Any-time algorithms are described there as algorithms which always present a solution when interrupted. The more time is invested in this algorithm the better is the resulting solution. In the DT planning context this means that the best current solution is returned.

The search algorithm currently used is a depth-first search of Prolog implemented by the resolution strategy. A complete action sequence to a given horizon is generated and saved to compare its value to other generated action sequences. To create an any-time algorithm there are two possibilities to modify the above search algorithm. The first possibility is to implement a breadth first search (BFS). But following this method one sees rapidly the drawbacks of BFS: it consumes much more memory than depth-first search. In general, the memory consumption of depth-first search is linear in the size of the solution. Breadth-first search consumes memory exponentially in the size of the solution. Even small tasks in the grid world with a horizon of five cause memory stack overflows when applied to a machine with 1 GB of memory.

The other possibility is to tweak the decision-theoretic optimization search of READYLOG such that it is able to handle a given time-bound. The idea we applied is to use iterative deepening depth-first search (IDS). It combines the benefits of depth-first search and breadth-first search. Its memory requirements are linear in the size of the given depth and the branching factor of the problem. It is also complete in cases where the branching factor is finite. Iterative deepening may seem wanton first, because same states are generated multiple times. Surprisingly, this fact is nearly negligible. Actually iterative deepening is *faster* than breadth-first search where several nodes of the next search level are generated. Iterative deepening does ignore the next level at first.

Another important point to be mentioned is the average branching factor of an arbitrary READYLOG program. For example, the branching of a nondeterministic choice is not only up to specified program which is to be solved, it is also influenced by the stochastic outcomes of all actions occurring in the program. In the grid world domain, for example, a program of the form $(up \,|\, down \,|\, left \,|\, right)$ has a branching factor of 16 because each action has 4 outcomes.

The procedure for solving a decision-theoretic problem was given as $\mathbf{solve}(horizon, \sigma, f_{reward})$ where σ is an arbitrary READYLOG program, $horizon$ is the number of actions to apply in one branch, and the f_{reward} is an arbitrary READYLOG function which assigns numerical values to states. This procedure was reformulated to fit the any-time idea by defining a new procedure $\mathbf{solve}(time, \sigma, f_{reward})$ where the $horizon$ argument was exchanged with a $time$ representing a time in seconds how long the calculations may last. Iterative deepening is started to search for a policy. If the time bound was or is going to be violated, the best policy found so far is returned.

The advantage of this approach is straight-forward. Instead of giving the program a fixed horizon on a computer with unknown capabilities, the programmer is able to define a time frame for the program. After this time the best policy generated up to this time point is returned and executed. Most importantly this is independent of the underlying hardware or READYLOG implementation. In cases of UNREAL or ROBOCUP (see Chapter 5) where fast decisions are necessary this is a major improvement to adapt to the dynamics of the game by adjusting the length of the decision cycle.

4.4 The Readylog Interpreter

In the following we get into details of the Prolog implementation of the READYLOG interpreter. First, we briefly show the domain axiomatization in Prolog, how the top-level loop of READYLOG works on a program, and describe how actions are executed within the framework. We describe how exogenous and sensing actions are integrated, show how fluents are evaluated applying regression, and give details about the implementation of passive sensing and action registers. At the end of this section we show the progression method integrated into the framework. The implementation follows a method proposed by (Lin and Reiter 1997). We assume that the reader is, in general, familiar with Prolog. Particularities of the Prolog implementation ECLiPSe (Apt and Wallace 2006) we use are indicated and explained in greater detail where needed. As usual we use the '!' as a symbol for a cut. We write conditionals as '$a \rightarrow b; c$'.

Domain Description in Brief

A domain description consists of the definition of primitive, exogenous, and sensing actions together with their effect axioms for each action. Further, fluents with their initial value have to be specified. These are implemented by the following set of facts:

$$\mathtt{prim\_action}(\mathtt{action}). \tag{4.8}$$
$$\mathtt{poss}(\mathtt{action}, \mathtt{condition}). \tag{4.9}$$
$$\mathtt{causes\_val}(\mathtt{action}, \mathtt{fluent}, \mathtt{value}, \mathtt{condition}). \tag{4.10}$$
$$\mathtt{senses\_action}(\mathtt{action}, \mathtt{fluent}). \tag{4.11}$$
$$\mathtt{exog\_action}(\mathtt{action}). \tag{4.12}$$
$$\mathtt{prim\_fluent}(\mathtt{fluent}). \tag{4.13}$$
$$\mathtt{initial\_val}(\mathtt{fluent}, \mathtt{value}). \tag{4.14}$$
$$\mathtt{exog\_fluent}(\mathtt{fluent}). \tag{4.15}$$
$$\mathtt{function}(\mathtt{functor}, \mathtt{value}, \mathtt{condition}). \tag{4.16}$$

In the above facts, `action` stands for an action term, possibly with arguments, `condition` for logical formula, `value` describes a formula which specifies the value of a fluent. For exogenous fluents one has to specify a clause which describes how the exogenous value can be achieved. This basically describes besides the agent program what has to be specified. What is missing is the connection to the execution system. In the following we will also describe these interfaces in detail.

The Mainloop

The mainloop of the READYLOG interpreter is implemented by a predicate $\mathtt{icpgo}(\mathtt{E}, \mathtt{H})$, where $\mathtt{E}$ is a READYLOG program and $\mathtt{H}$ is the current action history. Initially, $\mathtt{H} = \mathtt{s_0}$. The interpreter works on $\mathtt{E}$ in the following way, distinguishing between four cases:

1. *Exogenous Actions.* It is checked whether an exogenous action in the exogenous action queue which we describe below exists. If so, this action is inserted in front of the action history which means that it will be the next action to be performed in Step 2. Suppose the action queue before this step is $\mathtt{H} = [\mathtt{a_n}, \dots, \mathtt{a_1}, \mathtt{s_0}]$ and the exogenous action e occurs. The new action history then is $\mathtt{H_1} = [\mathtt{e}, \mathtt{a_n}, \dots, \mathtt{a_1}, \mathtt{s_0}]$. The predicate `exog_occurs` is defined below and checks if there is a new exogenous action in the queue. The `exog_action` predicate checks if the action `Act` is defined as an exogenous action. If not, the clause fails and the next clause of `icpgo` is tried, otherwise the action is inserted into the action history as described above. Finally, $\mathtt{icpgo}(\mathtt{E}, \mathtt{H_1})$ is called recursively, and the interpreter goes on with Step 2. The ordering of the `icpgo` predicates in the Prolog implementation ensures that in the next step a transition is taken, if possible.

$$\begin{array}{l}\mathtt{icpgo}(\mathtt{E}, \mathtt{H}) :- \\ \quad \mathtt{exog\_occurrs}(\mathtt{Act}, \mathtt{H}), \mathtt{exog\_action}(\mathtt{Act}), \mathtt{H_1} = [\mathtt{Act}|\mathtt{H}], !, \mathtt{icpgo}(\mathtt{E}, \mathtt{H_1}).\end{array}$$

2. *Transition of the Program.* A transition on program `E` is performed calling the `trans` predicate resulting in the transformed program E_1 on the action history H_1. The action a is executed by calling a predicate `icpxeq`, which we describe below. Finally, the predicate `icpgo`(E_1, H_1) is called.

```
icpgo(E, H) :−
    trans(E, H, E1, H1, P), icpxeq(H, H1, H2), !, icpgo(E1, H2).
```

3. *Final Condition Check.* At this step it is checked, whether a final configuration of the program `E` is reached by checking the predicate `final(E, H)`. The interpreter terminates if the `final` predicate evaluates to true on E and H.

```
icpgo(E, H) :− final(E, H).
```

4. *Waiting for Exogenous Events.* If no exogenous event occurred, no further transition of the program was possible, and the program is not in a final configuration, the interpreter waits for an exogenous action to become available from outside. The interpreter blocks until it receives a signal from the execution sub-system that a pending action command is completed.

```
icpgo(E, H) :−
    printf("icpgo(4) : WAITING FOR EXOGENOUS ACTION ...", [ ]),
    wait_for_exog_occurs, !, icpgo(E, H).
```

The predicate `wait_for_exog_occurs` is defined below.

Executing Actions with and without Sensing, and the Execution Transformer

Step 2 of the top-level interpreter predicate `icpgo` mentions the predicate `icpxeq`. This predicate defines the interface to executing action in the real world.

There are three cases to distinguish:

1. *No action* which changed the action history occurred. This can be seen as executing the nil action.

```
icpxeq(H, H, H).
```

2. *Primitive Action.* A primitive action is performed by an appropriate call to the low-level control, which is interfaced by the predicate `execute(Act, _, H)`. The second argument of this predicate is ignored, because the action was not a sensing action.

```
icpxeq(H, [Act|H], H1) :− not senses(Act, _), execute(Act, _, H), H1 = [Act|H].
```

3. *Active Sensing Action.* If the currently performed action is an active sensing action, the sensing result is inserted into the action history $\mathtt{H}_1 = [\mathtt{e}(\mathtt{F}, \mathtt{v}), \ldots]$, where F is the fluent which is associated to the sensing action, and v its respective sensing result. Note that this is exactly implements the sensing action histories from INDIGOLOG as described in Chapter 3.3.3.

$$\begin{aligned}&\mathtt{icqxeq}(\mathtt{H}, [\mathtt{Act}|\mathtt{H}], \mathtt{H}_1) \text{ :- }\\&\quad \mathtt{senses}(\mathtt{Act}, \mathtt{F}), \mathtt{execute}(\mathtt{Act}, \mathtt{Sr}, \mathtt{H}), \mathtt{H}_1 = [\mathtt{e}(\mathtt{F}, \mathtt{Sr}), \mathtt{Act}|\mathtt{H}].\end{aligned}$$

The predicate `execute` is the interface to the on-line execution system. With the predicate $\mathtt{holds}(\mathtt{online} = \mathtt{true}, \mathtt{H})$ it is evaluated if the built-in fluent $online$ in the current situation H holds. If so, the predicate `xTra` is called which will transform the action A into low-level commands of the robot or agent, otherwise the predicate evaluates to true, which means that the execution of the action is ignored. This will be the case, when the interpreter is in an off-line projection mode where no actions are performed in the real world.

$$\begin{aligned}&\mathtt{execute}(\mathtt{A}, \_\mathtt{Sense}\_\mathtt{val}, \mathtt{H}) \text{ :- } !,\\&\qquad \mathtt{holds}(\mathtt{online} = \mathtt{true}, \mathtt{H}) \rightarrow \mathtt{xTra}(\mathtt{A}, \mathtt{H}); \mathtt{true}.\end{aligned} \tag{4.17}$$

The case of executing actions on-line, we call another predicate `xTra` which finally interfaces the Prolog system with the rest of the low-level system that controls the robot or agent. In Chapter 6 we will discuss our robot low-level control system in detail. For now, it is sufficient to know that another system which takes care of executing the action in the real world, or gather sensor information through its sensors exists. xTra stands for “execution transformer” and performs for appropriate low-level system calls. These have to be specified for the particular application domain. But to give a better understanding of what we mean with interfacing the low-level system we give an example from the soccer domain, which we will describe in the next section in detail. In Section 4.2.2 we discussed the use of action registers. These mean that actions are sent to the low-level system with a special $send$ action, and the high-level controller will wait until it receives a message from the low-level process if it accomplished its task. The following example of an `xTra` predicate refers to such a send action of an action register called $nextSkill$:

$$\begin{aligned}&\mathtt{xTra}(\mathtt{send}(\mathtt{nextSkill}(\mathtt{PlayerNumber}), \mathtt{Skill}), \mathtt{H}) \text{ :- }\\&\quad (\mathtt{holds}(\mathtt{ownNumber} = \mathtt{PlayerNumber}, \mathtt{H}) \rightarrow \mathtt{set\_actuator\_skill}(\mathtt{Skill});\\&\qquad \mathtt{exoEnQueue}(\mathtt{reply}(\mathtt{nextSkill}(\mathtt{PlayerNumber}), \mathtt{nil}))), !.\end{aligned} \tag{4.18}$$

If the action sent via the action register is meant for the agent itself (denoted by $ownNumber = PlayerNumber$), the action is sent to the actuators, otherwise a reply message as described in Section 4.2.2 is sent back. Note that this reply message is scheduled as an exogenous event (with the `exoEnQueue` predicate). As we will lay out in our application examples the distinction between actions meant for the agent itself or for other agents gives an easy approach for executing multi-agent plans. Next, we discuss the integration of exogenous actions.

Exogenous Actions, Passive Sensing and Action Register

Other than primitive actions exogenous actions are not under control of the agent. They are imposed by the environment. Nevertheless, to be able to handle these kind of events, the axiomatizer needs for define all such events the agent should be able to react on. To store exogenous actions until they are processed, an action queue is introduced into the interpreter.

Step 1 of the top-level predicate `icpgo` looks for new exogenous actions via `exog_occurs`. The current available exogenous actions is popped from the queue. Note that for implementing action queues the non-logical predicates `assert` (which stores a current action in the queue) and `retract` (which deletes the current action from the queue) are needed.

```
exog_occurs(A, H) :- getExoAction(A, H).
getExoAction(A) :- exoDeQueue(A).
exoEnQueue(A) :- assert(exoQueue(A)).
exoDeQueue(A) :- exoQueue(A), retract(exoQueue(A)).
```

In Clause 4 of the top-level predicate `icpgo` the case when all other calls to `icpgo` failed, is handled. The behavior is to wait until an exogenous event occurs. In a non-blocking fashion it is checked if the predicate `exoQueue` succeeds, i.e. a new exogenous action was enqueued in the meantime. The non-blocking behavior is implemented by a loop over a sleep command. Note that repeat loops in Prolog are evaluated such that if the predicate in the loop body fails a backtracking up to the `repeat` statement is initiated. Thus, if the last predicate `exoQueue(Act)` succeeds the loop terminates, and with it the predicate `wait_for_exog_occurs`, and in turn Clause 4 of `icpgo`, which then leads to another call of `icpgo`.

```
wait_for_exog_occurs :- cylcetime(CycleTime),
    (exoQueue(Act); repeat, sleep(CycleTime), exoQueue(Act)), !.
```

Next we address passive sensing. Passive sensing is the process of updating the agent's world model in the background without performing active sensing actions. Passive sensing can be implemented making use of ECLiPSe Prolog's event queues. Their meaning is that the Prolog engine schedules goals within a given time interval. The clause `event` is scheduled after the time specified with `cycletime` by making use of the Prolog predicate `event_after_every(event, cycletime)`. Unlike hardware interrupts the interval is not fixed but the event is scheduled as an intermediate goal in the backtrack tree. This means that the clause `event` is called after `cycletime` if additionally the current intermediate goal in the backtrack tree has succeeded. We connect the event with the exogenous update action `exogfUpdate` which we described in Section 4.2.2. As the update action is an exogenous event it is inserted into the action queue in Case 1 of the main loop above and executed with the next call to the `trans` in Clause 2 of `icpgo`. This yields the world model update. Note that during off-line projections world model updates are not necessary and wanted. This behavior is implicitly given by the fact that during off-line projections exogenous actions are not executed (the fluent value of *online* is set to false and thus `xTra` is not called in Clause 4.17), which means that the update action is inserted into the action history but not executed in real.

Evaluating Fluents and Regression

In previous examples we several times used the predicate $\mathtt{holds}(\mathtt{F},\mathtt{H})$, which evaluates if a fluent F in the current situation H holds. In the semantic definition of a test action this exactly refers to the formula $\varphi[s]$ for test actions or loop conditions (cf. Section 4.1). What is needed to evaluate the predicate holds is regression to evaluate the fluent.[15] First, the fluent term has to be substituted by the value which it takes in the current situation.

To substitute terms the predicate subf/3 exists. It substitutes fluent terms by their value or evaluates simple arithmetic formulas in tests. We therefore test if the term to be substituted is a variable, a number, or a constant by the three clauses $\mathtt{subf}(\mathtt{P},\mathtt{P},\_\mathtt{H}) :- !,\mathtt{var}(\mathtt{P})$, $\mathtt{subf}(\mathtt{P},\mathtt{P},\_\mathtt{H}) :- !,\mathtt{number}(\mathtt{P})$, and $\mathtt{subf}(\mathtt{P},\mathtt{P},\_\mathtt{H}) :- \mathtt{const}(\mathtt{P})$. Simple arithmetic expressions are substituted with

$$\mathtt{subf}(\mathtt{A}+\mathtt{B},\mathtt{C},\mathtt{H}) :- !,\mathtt{subf\_arithm}(\mathtt{A}+\mathtt{B},\mathtt{C},\mathtt{H}).$$
$$\mathtt{subf}(\mathtt{A}-\mathtt{B},\mathtt{C},\mathtt{H}) :- !,\mathtt{subf\_arithm}(\mathtt{A}-\mathtt{B},\mathtt{C},\mathtt{H}).$$
$$\mathtt{subf}(\mathtt{A}*\mathtt{B},\mathtt{C},\mathtt{H}) :- !,\mathtt{subf\_arithm}(\mathtt{A}*\mathtt{B},\mathtt{C},\mathtt{H}).$$
$$\mathtt{subf}(\mathtt{A}/\mathtt{B},\mathtt{C},\mathtt{H}) :- !,\mathtt{subf\_arithm}(\mathtt{A}/\mathtt{B},\mathtt{C},\mathtt{H}).$$

We omit the somewhat lengthy definition of the subf_arithm predicate here. In a nutshell, it checks the operands of the expression whether it is a *t-form*, or if it is a list, or numbers. In case of a *t-form*, i.e. the operand must be a continuous fluent, the fluent *start* is evaluated and inserted into the appropriate *t-function* given in the fluent definition. If the operands are lists over values, the arithmetic operation is conducted component-wise, and finally, in the case that the operands are numbers, the operation is conducted and the results are returned.

Next we regard the case that the first argument of subf is a fluent. This respective case is implemented by:

$$\begin{aligned}
&\mathtt{subf}(\mathtt{P}_1,\mathtt{P}_2,\mathtt{H}) :- \mathtt{fluent}(\mathtt{P}_1),!\\
&\quad(\mathtt{special\_fluent}(\mathtt{P}_1) \rightarrow \mathtt{P}_3=\mathtt{P}_1;\mathtt{P}_1 =..\ [\mathtt{F}|\mathtt{L1}],\mathtt{subfl}(\mathtt{L}_1,\mathtt{L}_2,\mathtt{H}),\mathtt{P}_3 =..\ [\mathtt{F}|\mathtt{L2}]),\\
&\quad(\mathtt{register}(\mathtt{P}_1) \rightarrow \mathtt{has\_val}(\mathtt{eval\_registers},\mathtt{E},\mathtt{H}),\\
&\qquad\mathtt{E} \rightarrow \mathtt{has\_val}(\mathtt{P}_3,\mathtt{P}_2,\mathtt{H});\mathtt{not}(\mathtt{E}) \rightarrow \mathtt{P}_2=\mathtt{P}_3);\mathtt{has\_val}(\mathtt{P}_3,\mathtt{P}_2,\mathtt{H}).
\end{aligned}$$

In our implementation some special fluents like the fluent *pproj* for probabilistic projections, or the fluent *ltp* which encodes the least time point needed for **waitFor** statements and continuous fluents exist. These fluents are not substituted by their value because this is done later in another case of regression. If the $\mathtt{P}_1$ is an ordinary fluent it is recursively substituted. Note that the operator $=..$ constructs a Prolog list of a predicate beginning with its functor followed by its arguments. With the predicate subfl, which we will not give here, the list of arguments which also may contain fluent terms is parsed and substituted. Finally, another case is checked, namely if a special fluent register is given with $\mathtt{P}_1$. If so, it is evaluated if the respective fluent register has been evaluated before on the situation H, and if the variable $\mathtt{P}_3$ is bound accordingly. This special treatment is needed because the fluent register may only be evaluated once in a given situation. Finally, as the fluent $\mathtt{P}_1$ now is a ground term with all parameters substituted by their values, the

[15]We come back to the predicate holds when speaking about the implementation of progression.

value of the fluent can be assigned with aid of the `has_val` predicate which we will describe next. Other clauses of `subf` treat ordinary functions (for example, the reward function needed for DT planning), or the special action $set(f, v)$ (cf. Section 4.2.4) which sets the fluent f to value v (here, the argument f and must be substituted before its value can be obtained by calling `has_val`).

The predicate `has_val/3` evaluates the value of a ground fluent term. There are again several cases to be distinguished. The first case considers the initial situation. With the clause `has_val(F, V, [s0]) :- initial_val(F, V)` the fluent `F` will be assigned the value `V` which was stated in the domain axiomatization for the situation S_0. For each fluent one has to define its initial value by the predicate `initial_val`, as described in the beginning of this section. The second case addresses progressed fluent values. A fluent will take its progressed value, encoded by the clause `has_val(F, V, [S]) : -current_val(F, V, S), !.` Progression is subject to the next section, and we discuss it in detail there. The third case deals with fluent values which must be evaluated with regression.

```
has_val(F, V, [Act|H]) :-
    (nonground(F) → Cut = fail; Cut = true), sets_val(Act, F, V, H), (Cut →!; true);
    has_val(F, V, H), not sets_val(Act, F, V1, H).
```

`has_val` recursively steps through the action history until S_0 is reached and then successively applies the effect of action Act with calling the predicate `sets_val` which we show next. Note that the construction with the cut is important to avoid several calls of `sets_val` when backtracking over the predicate in case the fluent has free variables. The disjunction is needed to distinguish between the cases that the current action's effect axiom does not mention the fluent `F`.

Now we come to the `sets_val` predicate. The normal case is that the effect axiom (axiomatized with `causes_val(Act, F, V`$_1$`, P)`, where `Act` is the current action, `F` the current fluent, `V`$_1$ the effect on fluent `F`, and `P` a side condition which states when this effect is applicable) is applied if the side condition `P` holds on the current action history. Note that the effect formula has also to be substituted by its value.

```
sets_val(Act, F, V, H) :- causes_val(Act, F, V1, P)
```

Of course, there are again several other cases that have to be considered. The clause for a sensing action is

```
sets_val(e(F, V1), F, V, H) : -!, subf(V1, V, H).
```

which simply sets the fluent `F` to the sensing result `V`$_1$. Similarly, the case of our special $set(f, v)$ action.

```
sets_val(set(F, V), F, V, _) : -!.
```

Finally, we must regard our special update action which allows to update the whole world model in the background.

```
sets_val(exogfUpdate, F, V, H) : -!,
    exog_fluent(F), exog_fluent_getValue(F, V, H).
```

The predicate `exog_fluent_getValue` is an interface describing how to achieve a value of a fluent declared as on-line fluent (`exog_fluent`). It is usually directly coupled with the world model. This concludes our description of the implementation in brief. One has to note that there are further predicates like `trans`, `final`, and `bestDo` which implement the formal semantics of READYLOG. This can be nearly literally derived from the logical description of the semantics of READYLOG given in Section 4.1. We therefore omit them here.

Progressing the Database

As we showed in Chapter 3.3 the basic mechanism to acquire fluent values is to regress the action history. Applying the right-hand sides of successor state axioms, formulas with complex situation terms are reduced to formulas only mentioning ground situation terms, and thus S_0.

In practice this means that each time the program accesses the value of a fluent, the fluent formula has to be regressed to the initial situation to be able to get the value of the fluent. The problem which arises here is that when action histories grow large this is no longer efficient. In real world applications many fluents have to be accessed. Consider a tour-guide robot which is on operation for several hours without restarting the high-level controller. The length of the action history might be in the order of hundreds of thousand of actions. One could imagine that then regression is no longer the right choice.

Another approach makes use of progression. With progression, the database is rolled forward with each new action. This means that each performed action is directly manifested with calculating a new initial database. The basic idea is to calculate the effect of the action currently performed for each fluent and store this value in a new S_0. Unfortunately, it was shown by Lin and Reiter (1997) that progression for the situation calculus in general is only definable in second-order logic. Though, for some special cases, there exists also a first-order definable progression. We will here present the progression for the sub-class of the so-called relative complete initial databases. The general result about second-order definable progression can be found in (Lin and Reiter 1997). A relatively complete initial database $\mathcal{D}_{S_0}$ consists of a set of situation independent sentences together with a set of sentences of the form $\forall\vec{x}.F(\vec{x},S_0) \equiv \Pi_F(\vec{x})$ for each fluent where $\Pi_F(\vec{x})$ is a situation independent formula whose free variables are among the $\vec{x}$, i.e. $\Pi_F(\vec{x})$ does not mention terms of sort situation in $\mathcal{D}_{S_0}$.

Theorem 4 (Theorem 3 from (Lin and Reiter 1997)) *Let $\mathcal{D}$ be an action theory with a relatively complete initial database $\mathcal{D}_{S_0}$, and let α be a ground action term such that $\mathcal{D} \models Poss(\alpha, S_0)$. Then the following set*

$$\begin{aligned}&\mathcal{D}_{una} \cup \{\varphi \mid \varphi \in \mathcal{D}_{S_0} \textit{ is situation independent } \}\cup\\ &\quad \{\forall\vec{x}.F(\vec{x},do(\alpha,S_0)) \equiv \Phi_F(\vec{x},\alpha,S_0)[S_0] \mid F \textit{ is a fluent } \}\end{aligned}$$

is a progression of $\mathcal{D}_{S_0}$ to S_α, where

1. *Φ_F is the right-hand side of a successor state axiom,*

2. *$\Phi_F(\vec{x},\alpha,S_0)[S_0]$ is the result of replacing, in $\Phi_F(\vec{x},\alpha,S_0)$, every occurrence of $F'(\vec{t},S_0)$ by $\Pi_{F'}(\vec{t})$, where $\Pi_{F'}$ is as in the corresponding axiom for F' in $\mathcal{D}_{S_0}$, and this replacement is performed for every fluent F' mentioned in $\Phi_F(\vec{x},\alpha,S_0)$.*

The theorem states that replacing the right-hand side of the successor state axiom instantiated in situation S_0 for each fluent yields a progression of the database. To illustrate the above theorem we show an example from (Lin and Reiter 1997).

Example (Example 5.1, Lin and Reiter (1997)) First, we introduce the educational database from (Reiter 1991). There are two fluents (1) $\mathit{enrolled}(st, \mathit{course}, s)$: student st is enrolled in the course course in situation s; (2) $\mathit{grade}(st, \mathit{course}, \mathit{grade}, s)$: the grade of st in course is grade in situation s. Further, there are two situation independent predicates $\mathit{prereq}(\mathit{pre}, \mathit{course})$: pre is a prerequisite course for course course, and $\mathit{better}(\mathit{grade1}, \mathit{grade2})$: grade $\mathit{grade1}$ is better than grade $\mathit{grade2}$. The possible database transactions are (1) $\mathit{register}(st, \mathit{course})$: register the student st in course course; (2) $\mathit{change}(st, \mathit{course}, \mathit{grade})$: change the grade of the student st in course course to grade grade; (3) $\mathit{drop}(st, \mathit{course})$: drop the student st from course course. $\mathcal{D}_{ss}$ consists of the following successor state axioms:

$$\begin{aligned}&\mathit{enrolled}(st, c, do(a, s)) \equiv a = \mathit{register}(st, c) \lor \mathit{enrolled}(st, c, s) \land a \neq \mathit{drop}(st, c)\\&\mathit{grade}(st, c, g, do(a, s)) \equiv a = \mathit{change}(st, c, g) \lor\\&\qquad \mathit{grade}(st, c, g, s) \land \neg\exists g'.g \neq g' \land a = \mathit{change}(st, c, g').\end{aligned}$$

$\mathcal{D}_{ap}$ consists of the following action precondition axioms:

$$\begin{aligned}&\mathit{Poss}(\mathit{register}(st, c), s) \equiv \forall pr.\mathit{prereq}(pr, c) \supset \exists g.(\mathit{grade}(st, pr, g, s) \land \mathit{better}(g, 50))\\&\mathit{Poss}(\mathit{change}(st, c, g), s) \equiv \mathit{true}\\&\mathit{Poss}(\mathit{drop}(st, c), s) \equiv \mathit{enrolled}(st, c, s)\end{aligned}$$

Suppose now that the initial database $\mathcal{D}_{S_0}$ consists of the following axioms:

$$\begin{aligned}&\mathit{John} \neq \mathit{Sue} \neq C100 \neq C200, \quad \mathit{better}(50, 70), \quad \mathit{prereq}(C100, C200),\\&\mathit{enrolled}(st, c, S_0) \equiv (st = \mathit{John} \land c = C100) \lor (st = \mathit{Sue} \land c = C200)\\&\mathit{grade}(st, c, g, S_0) \equiv st = \mathit{Sue} \land c = C100 \land g = 70\end{aligned}$$

$\mathcal{D}_{S_0}$ is relatively complete, and $\mathcal{D} \models \mathit{Poss}(\alpha, S_0)$, where $\alpha = \mathit{drop}(\mathit{John}, C100)$. From the axiom for enrolled in $\mathcal{D}_{S_0}$ we see that $\Pi_{\mathit{enrolled}}(st, c)$ is the formula:

$$(st = \mathit{John} \land c = C100) \lor (st = \mathit{Sue} \land c = C200).$$

Now from the successor state axioms for $\mathit{enrolled}$, we see that $\Phi_{\mathit{enrolled}}(st, c, a, s)$, the condition under which $\mathit{enrolled}(st, c, do(a, s))$ will be true, is the formula:

$$a = \mathit{register}(st, c) \lor (\mathit{enrolled}(st, c, s) \land a \neq \mathit{drop}(st, c)).$$

Therefore, $\Phi_{\mathit{enrolled}}(st, c, \alpha, S_0)[S_0]$ is the formula:

$$\begin{aligned}&\mathit{drop}(\mathit{John}, C100) = \mathit{register}(st, c) \lor ([(st = \mathit{John} \land c = C100) \lor\\&\qquad (st = \mathit{Sue} \land c = C200)] \land \mathit{drop}(\mathit{John}, C100) \neq \mathit{drop}(st, c)).\end{aligned}$$

By the unique names axioms in $\mathcal{D}_{una}$, this can be simplified to

$$((st = \mathit{John} \land c = C100) \lor (st = \mathit{Sue} \land c = C200)) \land (\mathit{John} \neq st \land c = C100).$$

By the unique names axioms in $\mathcal{D}_{S_0}$, this can be further simplified to

$$st = Sue \wedge c = C200.$$

Therefore we obtain the following axiom about $do(\alpha, S_0)$:

$$enrolled(st, c, do(\alpha, S_0)) \equiv st = Sue \wedge c = C200.$$

Similarly, we have:

$$grade(st, c, g, do(\alpha, S_0)) \equiv st = Sue \wedge c = C100 \wedge g = 70.$$

Therefore a progression to $do(drop(John, C100), S_0)$ is $\mathcal{D}_{una}$ together with the following sentences:

$$\begin{aligned}
&John \neq Sue \neq C100 \neq C200, \quad better(70, 50), \quad prereq(C100, C200),\\
&enrolled(st, c, do(\alpha, S_0)) \equiv st = Sue \wedge c = C200,\\
&grade(st, e, g, do(\alpha, S_0)) \equiv st = Sue \wedge C100 \wedge g = 70
\end{aligned}$$

■

Theorem 4 basically gives us a scheme how to progress relatively complete initial databases. Each occurrence of a fluent in the initial database is replaced by its value instantiated on the current situation. Thus, we have to restrict READYLOG's basic action theories to be relatively complete. Recall that relatively complete means that the right-hand side of a fluent formula is situation independent. This on the other hand means that the progressed database can be computed by successively substituting fluent formulas by their value instantiated on the current situation.

In the following we briefly sketch our implementation of the progression of a relatively complete initial database. It consists of four steps.

1. *Copy an instance of the database* $\mathcal{D}_{S_0}$
2. *Evaluate the effects of action* α
3. *Update changed fluent values on* S_α
4. *Restore changed and unchanged fluent values to* $\mathcal{D}_{S_\alpha}$.

For the first step, we make use of the following predicate in our implementation:

```
current_to_temp :- retract_all(temp_val(_, _, _)), !,
    (clause(current_val(F, V, H) :- B),
    assert(temp_val(F, V, H) : -B), fail; true), !.
```

First, all previously remaining temporary values are deleted from Prolog's internal database before all existing `current_val` clauses are stored as temporary values. The construction `fail; true` exploits Prolog's backtracking mechanism. With this we get all `current_val` clauses, when Prolog backtracks at the `fail` token. In the second step the effects of the current action is calculated. In READYLOG the effects of an action are specified by an effect axiom. With the predicate

```
causes_val(Action, Fluent, Value, Condition)
```

the effect is specified. The last argument `Condition` can be used to attach further conditions to the value assignment. With this it is possible to implement conditional effects of an action. It is particularly easy to establish all the effects of executing an action with these kind of effect axioms. All `causes_val` clauses have to be evaluated and the respective fluent values have to be stored. In a next step the changed fluent values have to be added to the new database $\mathcal{D}_{S_\alpha}$. Basically, all effects as encountered by `causes_val` predicates are asserted as `temp_val`'s to Prolog's database. In a final step, all temporary values, i.e. those which remained unchanged from S_0 and those which were changed by action α are now manifested in the new database $\mathcal{D}_{S_\alpha}$ by copying the clauses back to the database. To access the such progressed fluent values we have to add the clause

```
has_val(F, V, [S]) :- current_val(F, V, S), !.
```

as first `has_val` clause to the interpreter core. Now, every time a fluent value is accessed the actual progressed database is queried first.

For our student example the copy of the initial database is

```
temp_val(enrolled(john, c100), true, s0) :- true.
temp_val(enrolled(sue, c200), true, s0) :- true.
temp_val(grade(sue, c100, 70), true, s0) :- true.
```

The action `drop(john, c100)` has the effect axiom

```
causes_val(drop(john, c100), enrolled(john, c100), false, true).
```

and thus the fluent value for `enrolled(john, c100)` becomes `false`. In the last step we have to restore the database with the changed and unchanged values. Finally, the database looks like

```
current_val(enrolled(john, c100), false, s0) :- true.
current_val(enrolled(sue, c200), true, s0) :- true.
current_val(grade(sue, c100, 70), true, s0) :- true.
```

In our implementation domain objects which occur as fluent parameters are referenced by an index. Thus, internally `enrolled(john, c100)` is represented as, say, `enrolled(1, 100)`. We make use of the built-in finite domain library which assigns an integer domain to each object. For example, the player of a soccer team are referenced by their player number and have assigned the domain $\{1, \ldots, 11\}$. This has the advantage that formulas like $x \geq 5$ or $5 \leq x \leq 8$ can be evaluated against their integer domain. Similarly, formulas $not(x = 1)$ can be computed easily.

To evaluate a logical formula there exists a predicate `holds/2`, which is defined as

```
holds(and(P1, P2), H) :- !, holds(P1, H), holds(P2, H).
holds(or(P1, P2), H) :- !, (holds(P1, H); holds(P2, H)).
    ⋮
holds(P, H) : -
    P =.. [Op, A, B], (Op = (=); Op = (<); (Op = (>); (Op = (=<); Op = (>=))))), !,
    eval_comparison(false, Op, A, B, Expr, H),
      (Expr = and(_) → holds(Expr, H); call(Expr)).
holds(P, H) : -proc(P, _), !, subf(P, P1, H), proc(P1, P2), holds(P2, H).
holds(P, H) : -function(P, _, _), !, subf(P, P1, H), holds(P1,H).
    ⋮
holds(not(P), H) :- !, (nonground(P) → holds_not(P, H); not holds(P, H)).
```

Negations are pushed inside with the last clause above. Unlike other GOLOG implementations (cf. Chapter 3.3), we do not make use of Prolog's evaluation strategy "negation by finite failure" for nonground terms, as for value comparisons like $\neg x = 1$ the domain of the parameter has to be set appropriately. In this special case, the domain of x has to be set to $x \in D \backslash \{1\}$. `holds_not/2` pushes negations inside the comparison. Therefore, we have to provide a set of predicates of the form

```
holds_not(and(P1, P2), H) :- !, holds(or(not(P1), not(P2)), H).
```

Comparisons have to be treated in a special way. We realize this by the predicate `eval_comparison` which was already used in the definitions of `holds/2` above.

```
eval_comparison(Not, Op, A, B, Expr, H) :-
    subf(A, A1, H), subf(B, B1, H),
      ⋮
    (Not = not → Expr = not(Expr2); Expr = Expr2);
    (is_domain(A2); is_domain(B2)) → fd_op(Not, Op, Op2), Expr =.. [Op2, A2, B2];
    Expr2 =.. [Op, A2, B2], (Not = not → Expr = not(Expr2); Expr = Expr2))).
```

One parameter of this clause is the Boolean flag `Not` which can be handed over to distinguish between negated and non-negated comparisons. The interesting part of this predicate w.r.t. progression is the call to the predicate `fd_op/3`. It substitutes the operand given by the variable `Op` with an operand of the finite domain library which in turn evaluates the results of the operation on the index structures.

One has to remark that progression does not come for free. Depending on the structure of the used effect axiom it is worthwhile not to use progression each time an action was executed. For our soccer domain implementations we found out that using a hybrid model, i.e. using regression and progression side by side up to an action history of 20 actions, regression is faster. After 20 actions we progress the database to get a new S_0.

4.5 Discussion

In this chapter we laid out our proposal for the Golog-based action language READYLOG. As we stated before, it is an amalgamation of several existing GOLOG dialects into one framework. For real-world domains one definitely needs a notion of continuous change. As in the situation calculus the world evolves from situation to situation, i.e. not continuous, an appropriate extension had to be integrated. We used the account in (Grosskreutz and Lakemeyer 2001; Grosskreutz and Lakemeyer 2003) where continuous change is formalized in the situation calculus and the dialect CCGOLOG is proposed. Further, uncertainty is omnipresent in realistic domains and an expressive action language has to provide a means to deal with it. Here, we follow two approaches: (1) is the possibility to reason with probabilistic programs as proposed in PGOLOG (Grosskreutz 2000; Grosskreutz and Lakemeyer 2000b), and (2) stochastic action models in the decision-theoretic planning context. Originally, probabilistic action models were proposed in (Pinto et al. 2000) (see also (Reiter 2001)). In order to simplify the programming of the outcomes of stochastic actions, we modified their definition from deterministic actions which describe the outcome to restricted READYLOG programs. We showed that both approaches have the same expressiveness. We introduced decision-theoretic planning as proposed in (Boutilier et al. 2000) into our framework.

To satisfy the requirements of real-time domains we proposed a new on-line execution model for decision-theoretic policies. We already discussed a similar approach which is due to Soutchanski (2001). To detect when a policy is no longer applicable due to unpredicted changes in the dynamic environment, we proposed a simple but efficient method to monitor the execution of policies. Previously, the problem of execution monitoring in GOLOG was regarded, although not in the context of monitoring policies. Nevertheless, we want to discuss the approaches in De Giacomo et al. (1998) and Bjärland (1999) in the following briefly. De Giacomo et al. (1998) and Soutchanski (2003) (the latter is the extended version of the former) introduce traces of histories. A $trace : program \times history \rightarrow history$ is a function which maps a program and a history to histories. They define a predicate $TransEM : situation \times program \times history \times situation \times program \times history$ which extends the original $Trans$ predicate of the CONGOLOG transition semantics to also take histories. If there exists a legal program trace for the successor configuration they call a predicate $Monitor$ which monitors the execution. The monitor predicate checks whether there have been relevant discrepancies between the successor situation with and without sensing actions or exogenous events, which might have happened in the meantime. If there are such discrepancies a recover predicate is invoked. To make this work a postcondition describing the original goal of the program being monitored must be specified. With the postcondition the goal of the monitored program is known. Thus, for a recovery strategy a forward chaining planning algorithm in GOLOG is used to detect a recovery strategy. Bjärland (1999) extends the approach by (De Giacomo et al. 1998) by taking also modeling faults into account. It might be that a particular effect is expected but when executing the associated action, a not expected effect occurs, and no exogenous event has happened in between. Similarly, we detect discrepancies between the expected state, based on model assumptions of our (stochastic) actions during decision-theoretic planning and real execution. Our recovery strategy currently is to simply cancel the policy and start planning from first principles. Given our time constraints this seems to be reasonable against the background that finding recovery strategy has the same complexity in general as planning from scratch. In Chapter 8 we develop some ideas how a recovery plan can be established, at least in the case where only little discrepancies are detected. Concluding this topic, there exist a body of work dealing with explaining execution failure. One example in the situation calculus is (Iwan 2002).

Finally, we introduced options (Hauskrecht et al. 1998; Precup et al. 1998; Sutton et al. 1999) into the READYLOG framework. These macro-actions lead to an exponential speed up in the planning times compared with using basic actions. Besides this speed up, the drawback of our approach is that our algorithm relies on explicit state space enumeration. This is a restriction, especially because the forward search algorithm used for calculating policies in READYLOG is not subject to such restrictions. In Chapter 7 we present some ideas how this can be overcome. Another restriction related to the state space enumeration is that for realistic domains our option approach is not directly applicable. One has to provide a clever state space abstraction for real-world applications such as robotic soccer. In Chapter 7 we present one possibility how the state space for the soccer domain could be abstracted making the option approach applicable. Some further commonly known concepts like caching techniques and pruning are presented in this chapter. They further decrease the computation times, but come at the cost of losing optimality.

Chapter 5

Simulated Readylog Agents

In this section we show application examples of the language READYLOG. We start with the interesting real-time dynamic domain of UNREAL TOURNAMENT 2004, an interactive computer game, which we will introduce in Section 5.1. We show how READYLOG can be applied for agents acting in this domain. We dwell especially on the trade-off between full agent programming and full decision-theoretic planning in the READYLOG framework. Then, we go over to the this domain. In Section 5.2, we first give an overview of this domain as this domain was central for the development of READYLOG. Here, we discuss the different ROBOCUP leagues and portray the different properties of this application domain. In Section 5.2.2 we concentrate on an example from the robotic soccer domain. We show how a double pass can be planned in ROBOCUP's Simulation league using probabilistic projections. We conclude with Section 5.3, where we discuss our approach as well as related approaches in the area of game AI and particularly, in the are of game bots.

5.1 Unreal Readylog Bots

In this section we will show a case study of decision-theoretic planning and modeling in the complex domain of the interactive computer game UNREAL TOURNAMENT 2004. We will compare different models how to pick up a sequence of "health item" needed to power up a READYLOG bot.

5.1.1 UNREAL TOURNAMENT 2004

UNREAL TOURNAMENT 2004 (Epic Games Inc. 2007) is a state-of-the-art interactive computer game. We have chosen this game because the bots available therein are programmed to behave like human adversaries for training purposes. The game engine itself is mainly written in C++ and cannot be modified. In contrast to this, the complete Unreal Script (in the following USCRIPT) code controlling the engine is publicly available and modifiable for each game. For instance, introducing new kinds of game play like playing soccer in teams or the game of Tetris have been implemented on the basis of the Unreal Engine (see (Epic Games Inc. 2007) for more information about these kind of 'game modes'). All this can be defined easily in USCRIPT, a simple, object-oriented, Java-like language which is also publicly available. Figure 5.1 shows a scene from an UNREAL TOURNAMENT 2004 game.

In UNREAL TOURNAMENT 2004 ten types of game-play or game modes have been implemented and published. For our work the following game types are of interest:

- *Deathmatch* (DM) is a game type where each player is on its own and struggles with all other competitors for winning the game. The goal of the game is to score points. Scoring points is done by disabling competitors and the secondary goal is not getting disabled oneself. If the player gets disabled he can choose to re-spawn[1] in a matter of seconds and start playing again. To be successful in this type of game one has to know the world, react quickly, and recognize the necessity to make a strategic withdrawal to recharge. An interesting sub-problem here is the games where only two players or bots compete against each other in much smaller arenas. In this setting one can compare the fitness of different agents easily.

- *Team Deathmatch* (TDM) is a special kind of Deathmatch where two teams compete against each other in winning the game with the same winning conditions as in Deathmatch. This is the most basic game type where team work is necessary to be successful. Protecting teammates or cooperating with them to disable competitors of the other team are examples of fundamental strategies.

- *Capture the Flag* (CTF) is a strategical type of game play. The game is played by two teams. Both teams try to hijack the flag of the other team to score points. Each flag is located in the team base. In this base the team members start playing. Scoring points is done by taking the opposing team's flag and touching the own base with it while the own flag is located there. If the own flag is not at the home base no scoring is possible and the flag has to be recaptured first. If a player is disabled while carrying the flag he drops it and if it is touched by a player of an opponent team, the flag is carried further to the opponents home base. If the flag is touched by a teammate who owns the flag it is teleported back to its base.

  To win such a game the players of a team have to cooperate, to delegate offensive or defensive tasks, and to communicate with each other. This game type is the first one which rewards strategic defense and coordinated offense maneuvers.

Note that the above game types include similar tasks. A bot being able to play Team Deathmatch has to be able to play Deathmatch just in case a one-on-one situation arises. Furthermore Capture the Flag depends on team play just like the Team Deathmatch.

[1]'Re-spawning' means the reappearance of a player or an item such that it becomes active again.

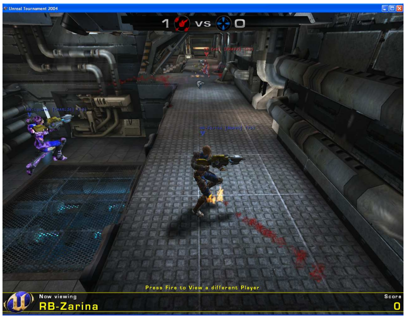


Figure 5.1: A scene from the interactive computer game UNREAL TOURNAMENT 2004.

5.1.2 Modeling UNREAL in READYLOG

The UNREAL bots are described by a variety of fluents which have to be considered while playing the game. All of the fluents have a time stamp associated such that the bot is able to know how old and how precise his state information are.

Identifier fluents: In the set of identifier fluents the bots name, the currently executed skill, together with unique IDs describing the bot and the skill can be found, among others.

Location fluents: The location fluents represent the bots location in a level, its current orientation, and its velocity.

Bot Parameter fluents: Health, armor, adrenaline, the currently available inventory in which the items are stored, and the explicit amount of each inventory slot is saved in this set of fluents. In the inventory additional game objective items can be found such as a flag in CTF.

Bot Visibility fluents: Here information about the objects in the view range of the agent are found. These information are distinguished in a team-mate and an opponent set. They contain the bots identifier and its current location. In games without team play the set of friends stays always empty during game-play.

Item Visibility fluents: Here the information about the currently visible and non visible items can be found. If an item is not visible at its expected position a competitor took it away and it reappears after a specific time. The definite re-spawn time of the item is unknown in general. The explicit re-spawn time is only available, if the bot itself took the item.

Bots in UNREAL TOURNAMENT 2004 are able to use the skills *stop, celebrate, moveto, roam, attack, charge, moveattack, retreat,* and *hunt*. All actions from UNREAL are modeled in the

READYLOG framework as stochastic actions and successor state axioms are defined for all the fluents.

To illustrate the capabilities of READYLOG we begin by showing two extreme ways to specify one task of the bot, collecting health items. One relies entirely on planning, where the agent has to figure out everything by itself, and the other on programming without any freedom for the agent to choose.

The example we use for describing the different approaches, their benefits, and their disadvantages is the collection of health items in an UNREAL level. Because collecting any one of them does not have a great effect the agent should try to collect as many as possible in an optimal fashion. Optimal means that the bot takes the optimal sequence which results in minimal time and maximal effect. Several constraints like the availability have to be taken into account.

The first and perhaps most intuitive example in specifying the collection of health packs is the small excerpt from a READYLOG program shown below. Using decision-theoretic planning alone, the agent is able to choose in which order to move to the items based upon the reward function. The search space is reduced by only taking those navigation nodes into account which contain a health item.

```
**proc** collect_naive
  ...
  getNavNodHealthList($HealthNodeList$) ;
  **solve**( $Horizon$, $healthReward$,
    **while** true **do**
      **pickBest** ($healthnode$, $NodeList$, moveto($own$, $healthnode$)))
    **endwhile**
  ...
**endproc**
**function** $healthReward$, $Reward$
  $CurrentTime = start \wedge$
  $TmpReward = CurrentHealth - CurrentTime \wedge$
  $Reward = \max(TmpReward, 0)$
**return** $Reward$
```

The first action of the excerpt binds the list of all health nodes in the current level to the free variable $HealthNodeList$. The $solve$ statement initiates the planning process. Up to the horizon $Horizon$ the loop is optimized using the reward function $f\_HealthReward$ which simply awards the health status of the bot discounted over the planning time. Note that we assume a continuously changing world during plan generation. The $pickBest$ statement projects the best sequence of $moveto$ actions for each possible ordering of health nodes. This results in the optimal action sequence given the bot's current location as health nodes which are far away are honored a lower reward.

Note that in this first basic example all calculations are up to the agent. Information about availability of items, the distance or the time the agent has to invest to get to the item become available to the agent as effects of the *moveto* action. While easy to formulate, the problem with this program is its execution time. With increasing horizon the computation time increases exponentially in the size of the horizon. All combinations of visiting the nodes are generated and all stochastic outcomes are evaluated. For example, in a setting with $Horizon = 3$ and $\#HealthNodes = 7$ the calculation of the optimal path from a specific position takes about 50 seconds,[2] which makes this program infeasible at present.

The next idea in modeling the health collection is to further restrict the search by using only a subset of all available health nodes. The example shown previously took all health navigation nodes of the whole map into account, whereas a restriction of those nodes is reasonable. Items which are far away are not of interest to the agent. Because of this restriction the real-time demands are fulfilled in a better way, but they are still not acceptable for the UNREAL domain. In the same setting as above ($Horizon = 3$ and $\#HealthNodes = 7$ from which only $Horizon + 1 = 4$ health navigation nodes are chosen) the calculation of the optimal path lasts about 8 seconds.

A much more efficient way to implement this action sequencing for arbitrary types is to program the search explicitly and not to use the underlying optimization framework. For example, filtering the available nodes and ordering them afterwards in an optimal way by hand is a much better way to perform on-line playing. First, the agent looks if there are item available with a rather high confidence value of 0.9. If no item with such a value come into sight, it is cross-checked whether there are item with a lower visibility value of 0.5. Finally, the agent starts to pick these item up. The example described above is illustrated in the program $\mathrm{collect}$ following next.

```
  proc collect( Type, Horizon )
    if $Horizon \neq 0$ then
        $Loc = botLocation$;
        getNextVisNavNode($Loc$, $Horizon$, $Type$, 0.9, $TmpVisList$)
    endif
    if $TmpVisList = \{\}$ then
      getNextVisNavNode($Loc$, $Horizon$, $Type$, 0.5, $TmpVisList$)
      $TmpVisList = [HeadElement|Tail]; NewHorizon = -1;$
    else $HeadElement = nil$ endif
  if $VisList \neq \{\}$ then moveto($HeadElement$)
  else collect($Type$, $NewHorizon$) endif
endproc
```

This example of modeling health collection is far from optimal from a decision-theoretic point of view. There are no backup actions available if something goes wrong and no projection of outcomes is applied during execution. On the other hand, the execution of the program $collect$ is computationally inexpensive. It therefore could be accepted if a sub-optimal solution is calculated.

Given the pros and cons of these two examples, it seems worthwhile to look for a middle ground. The idea is to allow for some planning besides programmed actions and to further abstract

[2]The experiments were carried out on a Pentium 4 PC with 1.7GHz and 1GB main memory.

the domain so that the search space becomes more manageable. Instead of modeling each detail for every action simpler models are introduced which do not need that much computational effort when planning.

To illustrate this we use an excerpt from our actual implementation of the *deathmatch agent* (program agent_dm below). Here an agent which chooses at each action choice point between the outcomes of a finite set of actions, was programmed. It has the choice between collecting a weapon, retreating to a health item and so on, based on a given reward function (see function $reward_{DM}$ on page 128).

The main part of the agent is the nondeterministic choice which represents the action the agent performs next. It has the choice between roaming, attacking an opponent, or collecting several specific items. The decision which action to take next is performed based on the reward of the resulting state. Note also that the non-deterministic choices are restricted by suitable conditions attached to each choice. This way many choices can be ruled out right away, which helps to prune the search space considerably.

```
proc agent_dm($Horizon$)
  while true do
    solve( $Horizon, reward_{DM}$,
      if $\neg sawOpponent$ then roam(own)
      else if $sawOpponent$ then moveattack(own)) endif
      ...
      | if $itemTypeAvailable(healthPack)$ then collect( $healthPack$ ) endif
      | if $\neg hasGoodWeapon \wedge itemTypeAvailable(weapon)$
      then collect ($weapon$) endif
    ) /*end solve*/
  endwhile
endproc
```

```
function $reward_{DM}$
  $\exists reward.(\ldots$
    if( $health < 150, reward_{health_1} = -1$),
    if( $health < 100, reward_{health_2} = -5$)
    ...
    if ($armor > 135, reward_{armor} = 20$)
    ...
    $reward_{score} = 200 * (max\_score - my\_score)$
    $reward = reward_{health_1} + reward_{health_2} + \cdots + reward_{score})$
return $reward$
```

Level Name	#Player	RB only	RB vs. UB
Training Day	2	9:6	8 : 9
Albatross	2	9:5	8 : 9
Albatross	4	9:8:5:3	8:1 : 9:5
Crash	2	8:7	7 : 8
Crash	4	9:7:5:3	8:5 : 9:6

Table 5.1: UNREAL deathmatch results generated in our framework. The setting was as follows: GoalScore = 9, LevelTime = 6 min, SkillLevel = skilled. We present the median result of five experiments for each entry here.

5.1.3 Experimental Results

In our implementation we connected READYLOG and UNREAL via a TCP connection for each game bot. With this connection the programs transmit all information about the world asynchronously to provide the bot with the latest world information and receive the action which the bot shall perform next until a new action is received. With this setup and after implementing an agent to play different styles of play, we conducted several experiments.

Perhaps the most important thing to be mentioned before attending to the explicit results is that the game is highly influenced by luck. Letting two original UNREAL bots compete in the game can result in a balanced game which is interesting to observe or in an unbalanced game where one bot is much more lucky than the others and wins unchallenged with healthy margin. Because of that we ran every test several times to substantiate our results.

Table 5.1 shows the results of the deathmatch agent which we described in the last section. In this and the next table the first column contains the name of the level we used for testing. The second column shows the total number of players competing in this level. In the following columns the results of different settings of the game are represented. The abbreviation UB stands for the original UNREAL bot. RB means the READYLOG bot. “RB only” means that only READYLOG bots competed. “RB vs. UB” means that the READYLOG bots compete against the UNREAL bots. The entries in line 3 and 5 mean the total ranking of the four competing bots, i.e. the winning bot got a score of 9, the second a score of 8, the third a score of 5, and the fourth a score of 3. In the column “RB vs. UB” the first two entries show the results of the READYLOG bots against the two UNREAL bots.

Next we consider the capture-the-flag agent, which was implemented based on the team deathmatch agent. Here we focused on the implementation of a multi-agent strategy to be able to play Capture the Flag on an acceptable level. We introduced two roles to implement a strategy for this type of game which we called *attacker* and *defender*. The attacker’s task is to try to catch the opponents flag and to hinder the opponents from building up their game. The defender’s task is to stay in the near vicinity of the own flag and to guard it. If the own flag is stolen its job is to retrieve it as fast as possible. Each role was implemented using a simple set of rules based on the

Level Name	#Players	RB only	RB vs. UB	Mixed
Joust	2	5:3	5:3	-
Maul	4	1:0	0:1	2:1
Face Classic	6	2:1	0:1	2:1

Table 5.2: UNREAL Capture the Flag results generated in our framework. The setting was as follows: GoalScore = 5, LevelTime = 6 min, SkillLevel = skilled. We present here the median result of five experiments for each entry.

state of each team's flag: the two flags can each be in three states, *at home*, *carried*, or *dropped*. For each of the resulting nine combinations of the two flags we implemented a small program for each role. For instance, if the own flag is in the state dropped the defender's task is to recapture it by touching the flag. For several states we introduced nondeterministic choices for the agent. It is able to choose between collecting several items or trying to do its role-related tasks.

The results can be interpreted as follows: In the one-on-one level *Joust* the READYLOG bot is surprisingly strong in game-play. We confirmed those results in other one-on-one levels. We think this is due to the goal directed behavior of our attacker. The agent does not care much about items and mainly fulfills its job to capture the flag and recapture the own flag. The column titled "Mixed" in Table 5.2 shows the result where READYLOG and UNREAL bots together made up a team.

5.2 Robotic Soccer

The next application example is from the robotic soccer domain. We show the action selection scheme of a simulated 2D soccer agent using READYLOG. As robotic soccer will be an issue also in Chapter 6 and in Chapter 7, we go more into the details and especially describe the domain properties which make the robotic soccer domain such a demanding and interesting domain. We start with our description of the domain before we go into the details of the READYLOG example.

5.2.1 The RoboCup Initiative

The RoboCup initiative was started in 1997 with the paper (Kitano et al. 1997). The idea was to foster AI and robotics research with proposing a common application domain: soccer. They provided the vision

> "*By the year 2050, [to] develop a team of fully autonomous humanoid robots that can win against the human soccer world champion team.*"
>
> (RoboCup 2006)

While this is clearly an ambitious goal, interesting insights about how to design and control autonomous humanoid robots can be made on the way towards this goal.

Several questions arise regarding the vision behind the RoboCup initiative. Why does one need a team of humanoid robots playing soccer? Why is it important to have a common test bed for research? Why soccer?

Soccer is of special interest as a common test bed because it has a very interesting characteristic: it is a cooperative and adversarial multi-agent/robot domain. There is a common goal (to win the game) which can only be achieved in a team, cooperatively. There are opponents that try to foil the own endeavors. We will discuss the domain aspects in detail below.

Even if one is not convinced that one needs a team of humanoid soccer playing robots the RoboCup idea seems to push the development towards the right direction. From a robotics perspective the interesting fields are how to build and control those robots or agents, including research on human gait and running, mobile power supply, control theory, engineering. From an AI perspective interesting fields are foundations of cooperative multi-agent systems, behavior programming, strategy acquisition, or decision making.

Trying to solve all problems at once will probably not lead to success. Therefore, RoboCup is organized in several different leagues which all concentrate on sub-problems.

Simulation League. The Simulation league, one of the first leagues in RoboCup, concentrates on agent research. It is, as the name says, a simulated league, where two teams of eleven software agent compete in a simulated environment. A simulation server, the Soccerserver (Noda et al. 1997), which receives the action commands from the agents and dispatches the sensory information to each agent exists. The Soccerserver calculates the visible information for each player and sends it in form of a string message to each agent via an UDP socket. The information of the visible landmarks of an agent are computed in an egocentric view. Besides the visual information the soccer server also sends aural messages to the player, i.e. player can shout and in a close distance around this can be heard by other players. The amount of data which can be sent via these "say" messages is restricted to 10 bytes per simulation cycle. From this information each agent has to construct a world model. Agents can settle actions by sending one of five basic actions back to the server. These actions are *dash*, *kick*, *turn*, *catch* (for the goal keeper only), *tackle*. The Soccerserver controls also the game play. An automated referee judges offsides, throw-ins and counts the goals. The simulation takes place in simulation cycles of 100 ms. That means that each agent can send an action to the Soccerserver every 100 ms. Visible and audible information are sent to the players every 150 ms. For the last three years a new simulation environment was established which extends the simulation to 3D (Obst and Rollmann 2004).

Small-size League. The Small-size league is a robotic league. Five small wheeled robots play on a field of the size of a table tennis board with a golf ball. As the robots are too small to carry sensors on-board, a ceiling camera is installed above the field. The camera images are sent to each team. Vision processing extracts the relevant information from the images. To alleviate the recognition, each player has a special color coding on top. With these information the actions the robots should perform are calculated by a computer off the field. The actions are sent back via radio to the robots. Thus, the league is partly autonomous. The research focus here is mainly on image processing and decision making.

Middle-size League. Two teams of up to five fully autonomous wheeled robots compete on a field of the size 8×12 m.[3] The robots may have a maximal size of 50×50 cm and the height may not exceed 80 cm. Like in the other soccer leagues are the goals and the ball color-coded to ease perception. The goals are painted blue and yellow, the ball is orange. The research focus of the Middle-size league is on robotics and on decision making. It turns out especially in this league that the whole system, hardware as well as the software, must form a unit. Only well integrated overall systems are competitive.

Four-legged League. While in the Small-size and the Middle-size league the hardware is developed by the participating teams and part of the research, the Four-legged league aims at developing robot control software on a restricted but common platform. The robots here are Aibo dog robots from Sony. The different developments and achievements made by the teams can be well compared, as they all work on the same platform. The robot's capabilities are limited. It has only a very small camera resolution, the sensor values of the joints are very noisy. Another problem in this league regarding the hardware platform is that Sony does not provide enough information about the hardware such that several controllers had the be reverse-engineered in order to lea rn how they work. Another remarkable note in this league is that it allows for distributed development. For example, there is a German Team (Röfer et al. 2004) where five universities work successfully together on the same code. Clearly, this is supported by the common hardware. Unfortunately, this league will sooner or later come to an end as Sony stopped the production of the Aibo in 2006.

Humanoid League. The ultimate goal of the RoboCup initiative is to play (robotic) soccer with humanoid robots. Of course, research on human-like robots must be conducted in order to achieve this goal. The humanoid league has existed for three years and makes remarkable progress. In the beginning, the competitions were only so-called technical challenges, where the teams showed the capabilities of their robots. Today, there are already soccer matches two-on-two. There are two different sub-leagues based on the sizes of robots, the so-called Kid-size league and the Teen-size league. In the Kid-size league, robots with a height from 30 cm to 60 cm compete, while a typical Teen-size robot measures between 65 cm and 130 cm, although in special cases robots up to 180 cm may participate in this league.

Rescue Leagues. Besides the soccer activities RoboCup has a broader scope. It turned out that the general idea to have competitions to foster certain research fields and aspects works very well. The Rescue leagues aim at rescuing entombed people from urban disaster areas, both in simulation

[3]The competition rules for 2007 are changed in that way that there are six robots on a field of size 12×20 m.

and with real robots. In the simulated leagues, research is about strategic planning, how resources like fire trucks has to be scheduled. In the hardware league, the partly autonomous robots must detect people. The problem is, for one, to detect the victims by the sensors, and for another that the robots have to maneuver in rough terrain.

RoboCup@Home League. The RoboCup@Home league was established in 2006. The idea here is to foster service robotics research. In a human apartment environment the robots should perform service robotics tasks. In the first competition the tasks were to safely maneuver through the apartment, follow a person, and pick up a newspaper. Further some free performance was allowed to show the capabilities of the robot. The focus of the several tests are the general applicability of the methods. Thus, each team has only five minutes time to adapt the environmental map of the robot to changes. Only the floor plan may be mapped in advance. The tests are conducted in such a way that first the developer may present the test to show the general applicability. In a second run the robot has to fulfill the task with a referee instructing the robot. An important aspect of the RoboCup@Home league is human machine interaction. Among other things, the robots should be able to understand natural language commands.

RoboCup Junior. Another important aspect of RoboCup is to interest students below university level for robotics. In the RoboCup Junior league students at high-school level program Lego Mindstorm robots for dance competitions or to fulfill simple tasks like follow a line on the floor.

A key aspect for research and education is the competition idea. In annual competitions researchers and students from all over the world get together. Participants can easily exchange experiences or research ideas. As the open source idea is widely spread with the RoboCup teams, it also possible not only to exchange ideas but also code. This helps a lot to bring the development further. Next, we will concentrate on the question why the soccer domain is an interesting research domain. In addition to the domain aspects discussed in Russel and Norvig (2003) we will concentrate on several other interesting properties of the soccer domain in the following and their impact to the system design of robots or agents acting in these domains.

The *dynamic and real-time aspect* of a domain means for one that the environment changes at any time unlike checkers which is round-based, for another that decision and actions must be taken in real-time. The decision cycles are short. The longer it takes to come to a decision the more the performance of the system decreases. The real-time aspect refers to nearly all system aspects. On the highest level this means that the decision which action is to be performed next has to be taken quickly. Consider the soccer domain where a player is in ball possession. If it takes too long to decide what to do next, an opponent might steal the ball, or the robot which is dribbling with the ball simply loses the ball because it stopped to think about when to kick the ball. This example shows that the real-time aspect of a domain is connected to several other aspects like acting in an adversarial domain. This aspect has an impact on the high-level decision layer: if it takes too long to decide, an opponent which decides faster will render the own efforts useless. If no opponents are involved there still are real-time aspects for the whole system. The example of dribbling a ball shows that motion control of the robot must also take decisions in real-time. If there are time latencies during execution, the robot will simply lose the ball while dribbling.

Many researchers intensively think about the term *physical embodiment*. For example, the Cognitive Robotics community discusses if a system acting in the real world must have a rigid body to give realistic results. Our experience over several years of doing RoboCup shows that it has significant impact on the design of decision making algorithms if the agent you deal with is physically embodied. Many good ideas turn out to be not feasible in the real world application. It also forces the system designer to meet the hard reality. Even in good simulated environments it is often not possible to generate results which are realistic enough to transfer the results directly to a real world application.

The soccer domain is only *partly observable*. The robot can only observe several aspects like its own position or the ball position. For instance, it may be that the robot cannot perceive what is happening behind it. For a simulated environment it means that not all important aspects are accessible.

Uncertainty is imposed by several other aspects of the robotic soccer domain. One reason for uncertainty for a soccer robot comes from the aspect of embodiment: we are dealing with “real” systems in the real world. This means that the actuators and the sensors of the robot are error-prone. The sensors are not accurate and thus impose uncertainty on the robot system. For example, consider the estimation of the position of the robot on the soccer field. The autonomous robot can only estimate its own location. The actuator system of the robot is imprecise and moreover coupled to the error-prone sensor systems. If the robot kicks the ball it can never predict exactly where the ball will be afterwards. The reason lies in the fact that we can only partly observe our domain. Many relevant aspects of the domain are not accessible to the agent. Again, consider the kick example. It will also depend on the pressure inside the ball where the ball will be after the kick. This information will never be available to the robot. In a simulated environment these problems are not ostensible. For this reason noise functions are used to simulate these effects also in simulated agent systems. Another kind of uncertainty is imposed by the fact that robotic soccer is an adversarial domain. The robot does not know the behaviors of its opponents. It can build models for the possible behaviors by observing them, but these form of prediction is also uncertain. The behaviors of the opponent should have direct influence on the decision making of the robot.

For many problems a winning *strategy* exists. One can prove that with this strategy the goal can be reached. For example, in checkers such a strategies is known. In a game like robotics soccer there definitely does not exist a global strategy which ensures to win the match. The term strategic domain also covers if there are strategies or tactics to achieve desired sub-goals of a game. Achieving these sub-goal do not necessarily ensure the achievement of the global goal, but might build a good base in doing so. For example, think of a defense strategy in robotic soccer. With a good defense which hinders the opponent team to shoot goals it is more likely to win a soccer match than without. The question is if such strategies for a domain exists, and which they are.

In domains like soccer or interactive computer games like UNREAL TOURNAMENT 2004 *adversaries* try to foil the endeavors of the agent to reach its goal. It introduces another source of uncertainty to the agent as it is hard to predict how the opponent might behave. It fosters the design of flexible and general approaches for achieving the desired goals of the domain. Another important aspect of an application domain for an agent system is the *cooperativeness*. This aspects is about if the global goal can be reached by one agent or of several agents need to cooperate in order to achieve the global goal. The aspect of cooperation has a major impact to the design of decision making. The design of the high-level control of an agent is also influenced if the domain is *partly episodic*. By this we mean that there are episodic elements in the application domain. For soccer this means that there are standard situations like free kicks or corner kicks. These situations define a subspace of all possible situations. For these episodes more specialized strategies can be developed.

After having introduced these domain properties of robotic soccer we now come to our first soccer application example. Throughout the rest of this thesis the soccer example will be our companion.

5.2.2 Action Selection in 2D Soccer Using Probabilistic Projections

Modeling a Double Pass

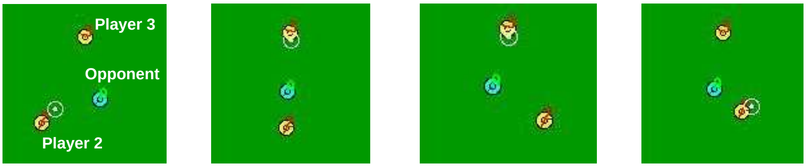
In the following we give an example of a READYLOG domain description for simulated soccer. In the subsequent programs the agent needs the primitive relational fluents $seeBall$ and $hasBall$, and the functional fluents $playMode$ and $passPartner$. The fluent $playMode$ is accessed via the exogenous action $changePlaymode$, which is triggered when the play mode in the Soccerserver changes. The update is done using the described passive sensing approach. The former fluent denotes the actual game state as broadcasted by the Soccerserver like *beforeKickOff* or *offSideLeft*, which is sensed by the agent. The latter takes the player number of a possible pass partner to play the double pass with. It will store the result of the probabilistic projection. Another continuous fluent $ballPosition$ is needed in the specification. To perform a double pass (see Figure 5.2) the agent will need the actions $search\_next\_passPartner$ and $directPass$. The first one is a sensing action which senses a player nearby that could be taken into account for playing a pass to. The $directPass$ action then performs the pass in the simulated environment.

The following action precondition can be modeled:

$$Poss(directPass(own, p)) \equiv hasBall(own) \wedge reachable(own, p)$$
$$Poss(search\_next\_passPartner \equiv true$$
$$Poss(changePlaymode) \equiv true$$

The effect for the pass action, for example, is that

$$Poss(directPass(own, p)) \supset \neg hasBall(own) \wedge hasBall(p).$$



(a) Beginning of the double pass
(b) Player 3 received first pass
(c) Player 2 moves to receive position
(d) Player 2 receives the second pass

Figure 5.2: Double pass scenario

The main control of the soccer agent can be modeled like given in the procedure $soccer\_agent$ below. The $withCtrl$ takes care that depending an the current play mode which is sent by

```
proc soccer_agent
  forever do
      withCtrl $playmode = beforeKickoff$ do place_on_field endwithCtrl
    || withCtrl $playmode = ownFreeKick$ do ... endwithCtrl
      ⋮
    || withCtrl $playmode = playOn$ do playSoccer endwithCtrl
  endforever
endproc
```

the Soccerserver the appropriate control procedure is exhibited. Each of these are interleaved concurrently with priority from top to bottom. Assume the current play mode is set to “play on”, i.e. normal play. Then the procedure *playSoccer* is called.

In this example we do not make use of decision-theoretic planning. We want to show how action selection can be conducted using solely probabilistic projections. So, the procedure calls in the following procedure refer to probabilistic programs. These programs either make use of the $prob$ statement or use probabilistic projections.[4]

We want to illustrate how a double pass can be selected. The procedure for probabilistically projecting a double pass is in the procedure $\mathrm{try\_double\_passes}$ below.

The action $search\_next\_passPartner$ senses the fluent $passPartner$ and sets it to the next possible pass partner. This information is retrieved from the world model. As long as there is a pass partner for the agent the loop condition remains true and the conditional is evaluated. A probabilistic projection over the procedure try_double_pass is initiated. If the probability of success for the projection is greater or equal a rather high value of 0.9, the double pass procedure is invoked, or otherwise the next pass partner which is offered from the world model is tested.

The probabilistic projection is performed over the try_double_pass procedure. In this procedure the agent who intends to play the pass (own) looks for a free space between itself and the

[4]Note that we do not make use of stochastic actions. All action models are deterministic ones.

```
proc try_double_passes
  search_next_passPartner;
    while $\neg passPartner = nil$ do
      if $pproj(has\_ball(own), try\_double\_pass(own, passPartner)) \geq 0.9$
        then set_next_action(double_pass($own$, $passPartner$))
        else search_next_passPartner($own$)
      endif
    endwhile
  endproc

proc try_double_pass($own$, $targetPlayer$)
  look_for_free_space($own$, $targetPlayer$);
  try_direct_Pass($own$, $targetPlayer$, $passNormal$) ;
  (receivePass($targetPlayer$) $||$
    intercept_direct_pass($closestOpponentToPass(targetPlayer)$, $targetPlayer$));
  if $ballKickable$ then
    kickTo($targetPlayer$, $freePos$, $0.8$) $||$
      intercept_direct_pass($closestOpponentToPass(own)$, $own$)
  else
    moveToPos($own$, $freePos$);
    receivePass($own$)
  endif
endproc
```

pass-receiving player ($targetPlayer$) for which the current projection is performed. This action sets the fluent $freePos$ to a position inside the free space behind the opponent player in order to offer a position where the second pass in this double pass scenario can be received. Then, the direct pass to the target player is performed in the simulation. Concurrently, the pass reception of the target player and a possible intercept action by the opponent player which should be outplayed is projected. When the next conditional is projected the world is in a state where the first pass was already played. Either, the target player received the pass in the projection, or the opponent player was able to intercept the ball. Therefore, it can decide if the ball is kickable for the target player. This condition is also used to separate the roles of both players in this double pass scenario. For the receiver the ball should be kickable, for the player who just played the first pass, the ball is no longer kickable. Thus, the target player kicks the ball back to the previously calculated position ($freePos$). Again, an opponent intercept action is taken into account. The player of the first pass starts running to the calculated receive position of the second pass and tries to receive the ball.

The model of the direct pass is given below. First, we project the effects of the direct pass action (by performing this action). Then, there are two possibilities for the further outcome of the pass. Either, the opponent which should be outplayed can intercept the ball (denoted by the $prob\_intercept\_direct\_pass$ procedure) or, concurrently, it is waited until the event that the ball is near the pass recipient or somewhere else has happened. With the $waitFor$ statement one respects the duration of the action as the real execution blocks until either condition has become true.

```
proc try_direct_pass($own$, $targetPlayer$)
  directPass($own$, $targetPlayer$, $passNormal$) ;
  prob_intercept_direct_pass($closestOppToPass(targetPlayer, targetPlayer)$)
  || waitFor($ball\_near\_player(targetPlayer) \lor ball\_far(targetPlayer)$)
endproc
```

For the modeling of the opponent behavior one could also easily integrate different models. In the following we show an example of how two different models for an opponent can be integrated. For example, $intercept\_direct\_pass_1$ can be a model describing a very sophisticated behavior of the opponent assuming the opponent to be very good at intercepting the pass. The other one might assume an opponent which is not so good at intercepting the ball. With the $prob$ statement the former gets assigned a probability of 0.7, the latter than has a probability of 0.3. These probability is taken into account when projecting over the pass.

```
proc prob_intercept_direct_pass($opp$, $targetPlayer$)
  prob($0.7$,
    intercept_direct_pass$_1$($opp$, $TargetPlayer$),
    intercept_direct_pass$_2$($opp$, $targetPlayer$))
endproc
```

Executing a Double Pass

To illustrate how READYLOG works in practice we give an example execution of a double pass. We refer to the procedures given above. We first introduce the example setting and show how a multi-agent plan is generated and executed in READYLOG by an execution trace of the program $\mathrm{try\_double\_passes}$.

Our scenario is the following. Player 2 (the lower yellow player in Figure 5.2 on p. 136) wants to outplay the opponent with a double pass. Player 3 (upper yellow player in Figure 5.2) is in a good position to play a double pass with Player 2. Player 2 therefore initiates the double pass by playing a direct pass to Player 3. Thereafter, Player 2 has to run to the position where it can receive the pass from Player 3 (Figure 5.2(b)). Player 3 receives the ball and should pass it back to Player 2 if Player 2 itself is near the reception position (Figure 5.2(c)). Finally, Player 2 receives the ball (Figure 5.2(d)).

To make it more concrete, we show the READYLOG execution trace of the described scenario in Figure 5.3. Although we are able to reason about the behaviors of opponents by appropriate models as well, we leave out this detail here. The left column of this figure shows the trace for Player 2 which initiates the pass, the right one for Player 3. Player 2 starts by getting a new world model and intercepting the ball to be able to play the first pass (lines 1 – 6). After the successful intercept action both agents start the procedure $try\_double\_passes(own)$. The variable own is set to Player 2 for both agents. Player 3 therefore plans all actions of the double pass from Player 2's perspective. Of course, in the execution each agent performs only actions regarding itself.[5]

The first action in this procedure is to find a pass partner. After resetting the fluent $passPartner$, the agents project all possible partners for playing a pass. This is expressed by the $pproj$ statement in procedure $\mathrm{try\_double\_passes}$ on page 137. This corresponds to the lines 7 to 13 in Figure 5.3. If this projection is successful with probability 0.9 the procedure $execute\_double\_pass$ is called. As one can observe in lines 14 and 15 in Figure 5.3, both players are executing the action `directPass`. The execution system can determine by the command `nextSkill`(2) that this action is for Player 2. Player 3 will not perform the action in the real world. We now enter phase 2 of our double pass (Figure 5.2(b)) where the first pass is to be received by Player 3. Again, both players settle on the same action ($\mathrm{receivePass}$). To synchronize the actions of both players the execution system waits until some condition meets denoting the end of the respective action. In our example the reception of the first pass is acknowledged by an exogenous event "*received pass*". (For ease of presentation we do not show this event as well as some control output in the programs code. They occur nevertheless in the execution trace. An example for this is indicated by the exogenous action in line 18 in Figure 5.3, `WAITING FOR EXOGENOUS ACTION`). The pass back from Player 3 to Player 2 is not modeled by a direct pass. Instead, a $kickTo$ action is performed to a position calculated by $look\_for\_free\_space$ in the procedure try_double_pass on page 137, i.e. a position in a free region behind the opponent. Note that the target position of the $kickTo$ command slightly differs in both traces. This can be explained by the different world models of the resp. agent based on which this calculation is done.

Figure 5.2(c) shows the situation when Player 2 is near the calculated receive position. Finally, Player 2 receives the pass (Figure 5.2(d)). As stated above, there can be small differences in values derived from the agent's world model. Therefore, to ensure that Player 2 receives the ball, it performs an intercept action in the end.

In this example we showed a multi-agent plan for a double pass. This plan does not use explicit communication to coordinate the agents involved. The execution of this plan is possible because both player reason about the same actions. Player 3 in the example generates the plan from the ball holder's point of view and comes to the same conclusion as Player 2. So, Player 3 identifies itself to be the best pass partner for Player 2. Multi-agent coordination like this only works if agents' world models are similar and not too uncertain.

5.3 Summary and Related Work

In this chapter we presented two application examples of READYLOG, the one was the simulated soccer example where the decision was taken based on probabilistic projections, the other was a READYLOG bot for the interactive computer game UNREAL. With the latter example of an opti-

[5]This is realized by the distinction between the own player and teammates in Clause 4.18 on p. 112 in Chapter 4.4. In the case the action was to be executed by a teammate, a reply action is immediately enqueued into the action queue.

```
                Player 2                                Player 3

send(getBasicWorldModel, true)
send(nextSkill(2), intercept)
WAITING FOR EXOGENOUS ACTIONS...
setPlayerProj(2,[-28.34,-2.33],...)
waitedIntercept
send(getBasicWorldModel, true)          send(getBasicWorldModel, true)
initializePassPartner                   initializePassPartner
setPassPartner(3)                       setPassPartner(3)
Prob. Proj. Test                        Prob. Proj. Test
 (# of initial configs: 1,                (# of initial configs: 1,
 unsorted/sorted # of traces:1/1).        unsorted/sorted # of traces:1/1).
Prob. Proj. Test (cached result).
write(PLANNED DOUBLE PASS.)             write(PLANNED DOUBLE PASS.)
send(nextSkill(2),                      send(nextSkill(2),
  [directPass, [3, pass_NORMAL]])         [directPass, [3, pass_NORMAL]])
setBallProj([-27.50,-3.88],...)         setBallProj([-28.50, -3.00], ...)
send(nextSkill(3), receivePass)         send(nextSkill(3), receivePass)
                                        WAITING FOR EXOGENOUS ACTIONS...
setBallProj([-23.27,-11.44],...)        setBallProj([-23.19, -11.43],...)
send(nextSkill(3),                      send(nextSkill(3),
  [kickTo, [[-18.71,-1.88],0.4])          [kickTo, [[-21.42, 0.98],0.4])
                                        WAITING FOR EXOGENOUS ACTIONS...
setBallProj([-27.50,-3.88],...)         setBallProj([-23.26, -8.75], ...)
send(nextSkill(2),                      send(nextSkill(2),
  [moveToPos,[[-18.71,-1.88],...])        [moveToPos,[[-21.42,0.98],...])
WAITING FOR EXOGENOUS ACTIONS...
setPlayerProj(2,[-18.71,-1.88],...)     setPlayerProj(2,[-21.42,0.98],...)
send(nextSkill(2), receivePass)         send(nextSkill(2), receivePass)
WAITING FOR EXOGENOUS ACTIONS...
setBallProj([-20.57,3.19],...)          setBallProj([-3.37, 2.97],...)
send(getBasicWorldModel, true)          send(getBasicWorldModel, true)
send(nextSkill(2), intercept)           send(nextSkill(2), intercept)
setPlayerProj(2,[-20.96,3.47],...)      setPlayerProj(2,[-23.30,-6.25],...)
waitedIntercept                         waitedIntercept
setPassFinished                         setPassFinished
setTrySucceeded(true)                   setTrySucceeded(true)
send(nextSkill(2), intercept)
WAITING FOR EXOGENOUS ACTIONS...
setPlayerProj(2,[-21.01,3.62],...)
waitedIntercept
```

Figure 5.3: Execution traces of the pass sender and receiver in the double pass situation

mal ordering of items for the item pickup tasks we showed the different modeling possibilities the READYLOG framework offers. One has to note that the 'best' model is dependent on the computational resources available. The framework for the READYLOG bot provides only the information the bot is able to see, that is, it has no complete world state. This is in contrast to the built-in bots, which can make use of the complete world state.

There have been other approaches to design AI-based game bots. Kaminka et al. (2002) implemented tasks like navigation, mapping, and exploration for game bots for the UNREAL TOURNA-

MENT framework. They used SOAR as the underlying agent framework in their approach. SOAR is general cognitive architecture for developing systems which exhibit intelligent behavior (Lewis 1999). Another example for the successful connection of SOAR and the UNREAL engine is given in (Magerko et al. 2004) where they implemented agents for the game Haunt II (Laird et al. 2002). The idea of this game is the following. Without being able to physically interact with the game world, the human player has to influence the AI-controlled bots in order to proceed in the game, that is, trying to scare the bots or to "possess" an AI character and manipulating its thinking processes. Another example is Munoz-Avila and Fisher (2004). Here the authors made use of HTN techniques to implement strategies for game bots in the UNREAL TOURNAMENT framework.

Buro is currently developing an open source real-time strategy game (Orts) which allows human players as well as machines to compete in a 'hack-free' environment (Buro 2003). Hack-free means that cheating is not possible. To ensure this only the server keeping all world information has a full world state (similar as the Soccer Server for the RoboCup Simulation league). Each cycle, all information available to a player (and only these information) are sent to her. In contrast to commercial games, this increases the amount of data processed and sent through the network each cycle, but the possibility of cheating is ruled out. Therefore this system is a well-suited platform for real-time strategy research. The server is still under development and, to the best of our knowledge, no clients have been developed yet.

Another closely related work is (v. Waveren 2001). v. Waveren describes the development of the Quake III Arena Bot. Quake III Arena is a fairly successful commercial game and like UNREAL TOURNAMENT 2004 it is purely based on multi-player gaming. The state based bots are implemented for training purposes and for gamers who do not possess an Internet connection. In (v. Waveren 2001) the architecture and methods used are described in detail, as are the problems which occurred during development. Recently, Bererton described the use of particle filters for state estimation in game AI (Bererton 2004). To reduce the agents' omniscience, they represent the knowledge of the bot by a computationally efficient method known from robotics. They simulate sensors for each bot and use the particle filter to estimate the state the bot is in. The example they use is the knowledge of the players' position in the environment. Instead of making use of the knowledge where the player is, the bot searches for him, estimates and tracks his position with the filter. This results in more realistic behaviours of the bots.

The work of Forbus et al. is about qualitative spatial reasoning to improve the AI of strategy games. In (Forbus et al. 2002) they describe the application of their ideas to computer games. They represent the environment in a qualitative fashion which results in better path finding, and more general strategy libraries. As examples they use military settings, e.g. ambushes or encircling manoeuvres. The already mentioned work of Funge (Funge 1998) is more GOLOG related. It describes the use of GOLOG in several simulated environments for computer animation, camera control, and physics-based applications situated in virtual worlds. Another work related to simulate computer controlled characters is that of Thurau et al. (Thurau et al. 2004) and (Bauckhage and Thurau 2004). They present approaches to behavior modelling by applying several machine learning and pattern recognition methods. They learn complex behaviors by observing the human player in how she is using the available elementary movements.

A good overview of AI techniques used in interactive computer games is given in the series *Game Programming Gems* (DeLoura et al. 2006) and *AI Wisdom* (Rabin 2003). New developments and applications are contributed to this series. These examples give an overview about other works in the field. The relation to our work is that they also apply AI techniques to increase the performance of game bots in various ways. Our results show that compared with the built-in game bots, our approach is competitive even with a restricted world model. The READYLOG bots can compete with the omniscient UNREAL TOURNAMENT bots, though, currently, we are not able to play out all the strength of READYLOG. These domains are demanding in that there exist very many items in a game level which have to be represented by fluents. Having too many fluents is a problem for both regression and progression in READYLOG. Clearly, one could apply qualitative abstraction techniques to downscale the number of item needed to be represented. These problems resulted in that only a small horizon for decision-theoretic planning was possible. But the examples from simulated soccer and interactive computer games showed the different possibilities of modeling the environment and the behavior of an READYLOG agent. The possibility to choose the level of behavior modeling between programming and full planning or probabilistic projections for selecting the most appropriate actions is one of the appealing features of READYLOG.

Chapter 6

Embodied Readylog Agents

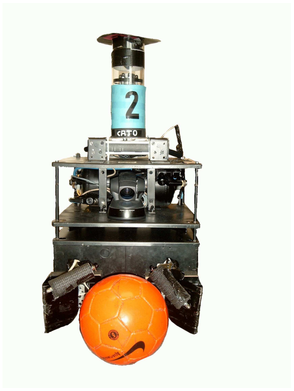
6.1 Introduction

As we pointed out in Chapter 5.2 is the embodiment of an agent system an important aspect also for high-level decision making. The abilities of a real robot, and especially the imperfection of its sensors and actuators together with imprecision resulting from small deviations on each layer of the control software which may cumulate, have an influence on the high-level control. For successful cognitive robotics applications one needs a control system which forms a union. A good reasoning component does not help if the robot has trouble driving to a given position or is not able to localize robustly, and vice-versa, a good low-level system does not lead to a robust museum tour-guide if the high-level component is not able to react appropriately to unforeseen changes in the world. This means that the control software must be tailored to the hardware system. In the following we sketch our hardware platform that was built with the aim to be able to compete in robotic soccer as well as to provide a stable ground for service robotics applications.

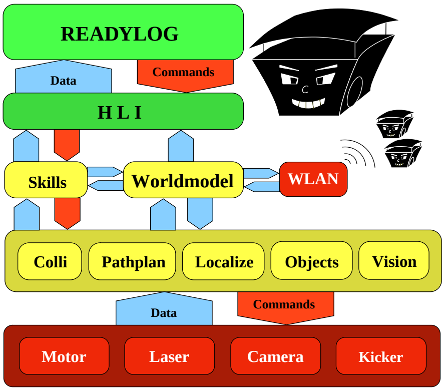
The hardware platform of the AllemaniACS Middle-Size league RoboCup Team has a size of 39 cm $\times$ 39 cm $\times$ 80 cm (Figure 6.15). The robot is powered by a differential drive, that is, the robot has two wheels that can be controlled separately, and another castor wheel. The differential drive allows the robot to turn in place. The advantage is that a differentially driven robot can very well drive in a straight line, but the disadvantage at least for the soccer application is, that it is non-holonomic, i.e. the drive poses some restrictions on the possible trajectories and the robot cannot move ad lib to all its degrees of freedom. A holonomic wheeled robot is able to move sideways, too.

The motors have a total power of 2.4 kW and are originally developed for electric wheel chairs. They provide the robot with a translational top speed of about 2.5 to 3 $\mathrm{m/sec}$ and a rotational speed of nearly 1000$^\circ/\mathrm{sec}$. The motor power is needed as the robot has a total weight of approximately 70 kg. For power supply there are two 12 V lead-gel accumulators with 15 Ah each on-board. The battery power lasts for approximately two hours at full charge.

As one can see in Figure 6.15 the robot has several layers. On the first layer above the base, a laser range finder (LRF) is installed. It scans a field of 360° with one degree resolution in



(a) AllemaniACs Robot



(b) Software Architecture

Figure 6.1: The "AllemaniACs" System

one sweep. The LRF provides measurements with a scan frequency of 10 Hz. This is our main sensor and is used for tasks like navigation, collision avoidance, and localization. We discuss our approaches to these tasks in detail throughout the next section. On the next layer a Sony EVI-D100P camera mounted on a pan/tilt unit is installed. The camera provides us with images in PAL resolution at a frequency of 25 Hz. The main task for this camera is to recognize objects. In the soccer application this is mainly the ball. Behind the camera, parts of the air tank for our pneumatic kicking device becomes visible. On top, the IEEE 802.11a/b/g access point for wireless communication and an omnicamera is mounted. The omnicamera is a low-cost development with a web cam which is pointed towards a hyperbolic mirror. This camera allows a $360°$ view field around the robot. The aim for this device is to roughly approximate the position of the orange color blob (the ball) in the image. The information from this camera should only give a hint where the ball is located, and therefore a very low-cost light bulb serving as the hyperbolic mirror is sufficient.

On-board the robot has two Pentium III PC's at 933 MHz running Linux, one equipped with a frame-grabber for the Sony EVI-D100P camera. This platform allows for soccer playing, but is suited also (and by now even better) for service robotics applications.

6.2 Robot Control Software

In this section we look at the low-level control software of the AllemaniACs robots. The system uses a classical three layered architecture with an interface layer between the hardware and the control modules on the middle layer, which in turn build the interface to our high-level decision making with READYLOG. The middle layer comprises modules like navigation, localization, or object recognition. While we have a detailed view on the tasks navigation, localization, and a comparative study of several sensor fusion techniques for merging the ball perception of the robots, we overview tasks like vision, path-planning, and object recognition only briefly. The third layer of the system architecture consists of the world model and the reasoning component READYLOG.

The software architecture is shown in Figure 6.1(b). As already mentioned the system consists of three layers. The control flow is as is usual in layered hierarchical architectures from bottom to top concerning data and from top to bottom w.r.t. control commands (cf. e.g. (Murphy 2000)).

Inter-process Communication

For communication between control modules we make use of a blackboard system. Each module connects to the blackboard system and is able to read data provided by other modules from the blackboard. Inside the blackboard several data sections are separated and access rights are regulated. To avoid dirty reads we make use of a client-server structure to govern read/write permissions. For each data section in the blackboard only one server process which may write to the respective data section, is allowed. This avoids dirty reads as several writers cannot overwrite data from others. Reader processes have no restriction, several modules may read a data section. For another, to assure atomic read and write operations semaphores are used for reading and writing. The blackboard is realized as a shared memory segment. At start up of a new module it connects to the blackboard and registers the data it is allowed to write, as well as the sections is wants to read data from. By using a shared memory for exchanging data on one host, one meets the real-time conditions closely, as there is basically no overhead in storing the data. There are no time guarantees for accessing the data and all modules connected to the blackboard are running asynchronously. The age of the data are tracked by timestamps. The consumer modules have to decide on their own if a date is too old for their application. Race conditions, though, are mostly avoided by guarding the memory segments with semaphores. For inter-host communication the blackboard offers the possibility to use TCP or UDP connections to communicate with remote blackboards. This is an important feature as our robots are equipped with two hosts over which the several control modules are spread.

Low-Level Interfaces

The low-level interfaces are basically hardware drivers with access to the blackboard. The *motor* driver provides data like odometry information which are calculated from the wheel encoders and estimates about the velocity of the robot on the one hand, on the other hand it takes driving commands from modules of upper levels. The *laser* driver takes commands for starting or stopping the LRF, and provides 360 distance measurements per sweep. As the *directed camera* is with its pan/tilt unit also an actuator it can take commands like $move(\varphi, \theta)$. It provides the vision module with camera images. The omnivision camera only yields raw images. Finally, we have the *kicker* interface which takes commands actuating the pressure valves of the pneumatic kicking device.

The Middle Layer

The modules on the middle layer work on the data provided by the sensors. A central task especially with fast heavy-weight robots is an effective *collision avoidance* strategy. We present our approach in Section 6.3 in detail. For successful complex robot operations in dynamic environments a working localization is needed. Here we make use of a Monte Carlo approach which we show in Section 6.4. To endow the robot's world model with a rich representation of the environment, one moreover needs a good object classification. Here we use the information provided by the vision module, and further we make use of the fact that the robot is localized on its given environment map. Thus it is able to distinguish between static and dynamic obstacles. The dynamic obstacles are classified by their laser signature. For the soccer application, detecting the ball is an important feature. The *vision* module inspects a camera image on several scan lines. If a sufficient number of pixels on a scan line has the appropriate object color we grow a region of interest around these pixels. Each region is then color segmented. The segmentation is based on a color map which is gained in a color calibration process, where the different colors are trained in a supervision mode on a few sample images. The thresholds for the different colors are found following a Bayesian approach. For finding the ball we apply a randomized circle fitting following (Chen and Chung 2001). The circle fitting is implemented as an any-time algorithm which returns the best fitted circle. With a geometric model of the robot the position of the ball is estimated.

Interfaces to Readylog

Above the middle layer Figure 6.1(b) shows the modules *world model*, and *skills*. These are the modules with which our high-level framework READYLOG is connected with. From the point of view of high-level decision making, the skill module encapsulates actuators, the world model has the same function w.r.t. sensor data. The skill module provides the basic actions for READYLOG. These are for example actions like *drive to global position* or *turn with angle θ* or more sophisticated ones like *dribble around opponents*. While the basic actions are clearly influenced by the soccer application, they are nonetheless useful for service robotics applications as well. In Section 6.6 we give a detailed overview of the different actions available. The world model accumulates all the available data from the sensor systems. This data comprises the location of the robot in the map, the locations of the teammates, opponents and the ball for the soccer application, and also tactical information are calculated and stored here. We come back to world modeling issues in Chapter 7. One can distinguish between local data which are derived from the perception of the robot, or global data which also makes use of information of the teammates that are communicated among the robots. When addressing sensor fusion in Section 6.5 we discuss this matter. While READYLOG is implemented in Prolog and the rest of the software is implemented in C++, we need another interface between READYLOG and the low-level control software. This function has the module *HLI*, the high-level interface. It translated Prolog calls to appropriate C++ function calls.

6.3 Laser-based Collision Avoidance on a Mobile Robot

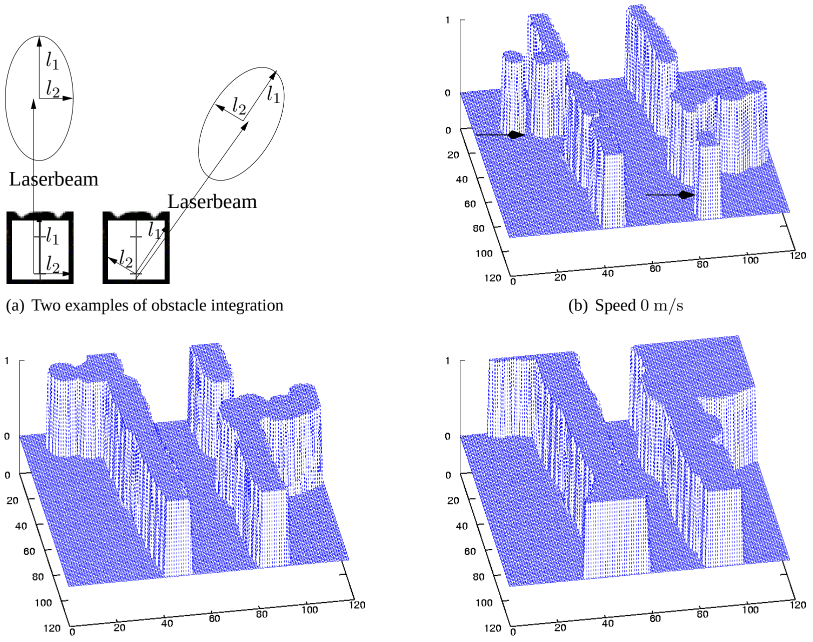
One central task for a mobile robot is to safely maneuver in its environment. Therefore, the robot must find a collision-free path to a given target location. The objective is to find such a collision-free path while maximizing the robot's speed. In dynamic environments the method applied must also account for obstacles which may suddenly appear in the robot's trajectory. The control algorithm must either be able to drive around the dynamic obstacle or, if this is not possible, it has to stop right in front of the obstacle without colliding.

In this section we describe our method for attacking the problem of collision avoidance. We make use of the 360° laser range finder which provides us with distance measurements in a frequency of 10 Hz. Roughly, the algorithm works as follows: first, a collision-free path to the target point is found by applying A^* search over an occupancy grid with a fixed size. The last visible point on the path lying in the grid is selected as a local target point. Ray-tracing with the current orientation of the robot yields a collision point in front of the robot, i.e. the point where the translational trajectory hits an obstacle. Using the target point, the collision point and the robot's location we construct a triangle. By the way we construct this triangle we ensure that every grid cell in it is collision-free. To achieve optimized velocities for approaching the local target point, A^* searches for $\langle a_{trans}, a_{rot}\rangle$-values which keeps the robot inside the triangle and which minimizes the time to reach the target point. In the following we describe the method in detail.

Navigation Algorithm

Our algorithm is based on an A^* search in the $\langle x, y\rangle$-space. The resulting path is modified in such a way that the robot can follow this path for as long as possible without the need to calculate a new one in each cycle of the algorithm.

```
**Input**: laser scan $s$, odometry position $p_r = (x_r, y_r), \theta_r$, target position $p_t = (x_t, y_t), \theta_t$,
        current velocity $v_r = (\upsilon_r, \omega_r)$
**Output**: motor commands $c = (\upsilon_m, \omega_m)$
path $P$ = $\{\}$, $P_{prev} = \{\}$
**while** $\neg atGoal(p_r, p_t)$ **do**
    $M$ = buildOccupancyMap$(s, p_r, p_t, v_r)$
    **if** valid$(P)$ **then** $P$ = $P_{prev}$
    **else** $P$ = searchPath$(M)$
    **end**
    **if** $|P|$ = 0 **then** $c$ = escapeAction
    **else**
        $P_{opt}$ = smoothPath$(P)$
        $c$ = calculateMotorCommand$(P_{opt})$
    **end**
    $P_{prev}$ = $P_{opt}$
**end**
```



(a) Two examples of obstacle integration

(b) Speed $0\,\mathrm{m/s}$

(c) Speed $1\,\mathrm{m/s}$

(d) Speed $2\,\mathrm{m/s}$

Figure 6.2: Examples of integrating obstacles in the occupancy grid map.

Each cycle, i.e. each time the navigation algorithm is called, it retrieves an up-to-date sweep from the laser range finder, the robot's own pose, which is derived from the odometry, the target pose and the robot's current velocity as its input. First, we integrate the distance measures into an occupancy grid and search for a path to the target point. If a path from a previous cycle of the algorithm exists, it is checked whether this path is still valid, i.e. if it is still collision-free. A new path is calculated otherwise. In the next step the path is smoothed. This means that in this extra computation step it is checked if the path could not be displaced further away from obstacles. This is especially useful in narrow doorways. Moreover, this makes the path more persistent, i.e. a once generated path will be used during subsequent cycles of the algorithm if possible which saves computation time. In a final step appropriate motor commands in terms of translational and rotational velocities are calculated. If no path could be found in the search which means that the robot is surrounded by obstacles, a special escape procedure tries to navigate the robot slowly out of the stuck situation. If the robot is not in an escape situation, the generated path is smoothed.

Building the Occupancy Grid Map. In the first step of our algorithm we integrate the laser sweep into an occupancy grid. In case of our soccer robots, we integrate 360 single values, for the B21 only 180.[1] The occupancy grid is the basis for our path calculation. At the beginning, the robot is located in the origin of the 2D occupancy grid representation. To ensure safe passage of the robot during the execution of the next driving command, we take account for the robot's velocity. The faster the robot moves the greater the security distance to the obstacles should be, or the farer away from obstacles the calculated path should be. Therefore, in the step of integrating new sensor information, we represent a single distance measurement from the laser range finder not as a single occupied cell, we calculate an ellipsis which depends on the size and the velocity of the robot. Figure 6.2(a) shows two examples. The length of the robot is l_1, the width of the robot is represented as l_2. Note that the width and the length of the robot (and therefore of the ellipse) also depend on the angle of the laser measurement. Further note that the ellipses in Figure 6.2(a) are calculated when the robot stands still, i.e. $v_r = 0$. In case the robot has a velocity v we multiply l_1 and l_2 with the velocities in x and y direction:

$$l_1 = l_1 + l_{sec} + l_1 \cdot |\sin\theta \cdot v|, \quad l_2 = l_2 + l_{sec} + l_2 \cdot |\cos\theta \cdot \omega|$$

with θ denoting the angle of the respective measurement, $\langle v, \omega \rangle$ the translational and rotational velocity of the robot, and l_{sec} an additional security distance.

The idea of integrating the velocity of the robot is the following: to account for the safety aspect that the robot has to be able to stop in front of an obstacle at any time the size of the obstacles is extended. Thus, the robot itself can be represented as a mass point which simplifies the detection of possible collisions between the robot and an obstacle in the algorithm.

We use a fixed occupancy grid size. Thus, we cut off sensor readings at a certain distance to keep the search for an optimal path in a certain time-bound. When the robot is moving at high speed, we shift the position of the robot in the occupancy grid so that it uses more look-ahead.

Figure 6.2 shows the integration of laser readings into the grid map. The laser scans were taken from our department hall. Note that we only applied translational speeds and the robot was oriented down the hallway. Therefore, the ellipses were only expanded in parallel to the robot. On the right wall the robot can "see" into an open door. The repositioning of the robot inside the grid with increasing speed can be observed by looking at the doorway on the right-hand side which moves "down" the hall in Figure 6.2(b)-6.2(c). Our soccer robots have blind spots where the mounting for the layer with the camera is installed (see the markings in Figure 6.2(b)). One can observe these with the loops in the hallway walls. With increasing speed these blind spots disappear since the obstacles are further expanded.

Searching a Path with A^*. After integrating the new laser distance measurements into the occupancy grid, we calculate an optimal path from the current position to the target position. If

[1] We also implemented this method on the RWI B21 robot Carl which is installed at our department.

the target position lies outside the grid range, then we project the target point onto the border of the grid to get a local target.

The heuristic used for the A^* search is the Manhattan distance between the robot and the target point. The possible actions in each state of the search are $A = \{N, NE, E, SE, S, ...\}$, i.e. all unoccupied neighboring cells that can be reached in one step of the search. The cost function between two neighboring grid cells i and j is defined as

$$cost(a,b) = \begin{cases} d & \text{if } i,j \text{ adjacent,} \\ \sqrt{2}d & \text{otherwise} \end{cases}$$

and $\forall k \exists j.cost(i,k) = cost(i,j) + cost(j,k)$. If the distance weight d is set to 1, the cost function describes the Euclidean distance of two grid cells (normalized by the cell size). There is no need to integrate a more complex cost function such as a function which decreases with the distance to an obstacle or is proportional to the occupancy of a grid cell as in (Stachniss and Burgard 2002) (which takes an extra value iteration step over the grid for each new target point)[2] because we can guarantee that the robot takes a security margin around the obstacle as described above. Moreover, the calculation of such a cost function is more expensive than using our simple cost function plus taking the extra computation time to integrate the obstacles as ellipses into the grid.

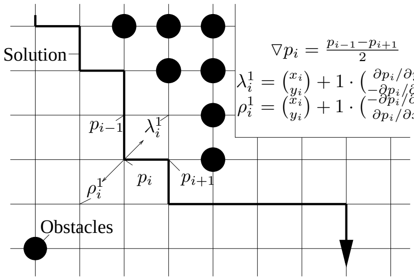
This step of the algorithm yields a path $P = \langle p_r, p_0, p_1, \ldots, p_t \rangle$, where $p_r = (x_r, y_r)$ is the robot position and p_t is the target position, which can either be the global target position or which is a local target position that resulted from the projection onto the border of the grid.

Smoothing the Path. For stable navigation it is important that the path does not change too much from cycle to cycle. The worst case would be if the path planning method returns a new path each cycle which could lead to oscillating behavior. The reason for this alternating solutions is mainly the sensor noise (even if a sensor model is integrated when calculating the grid) and the fact that in the step of integrating the laser distance measurements rounding errors might occur (due to the sine and cosine operations). The result is that the integrated obstacle positions might swing. To prevent recalculations of the whole path we stabilize the path by shifting it further away from obstacles nearby. This is important for narrow doorways. With smoothing the path as described below the robot is able to traverse narrow passages faster than using solely the planned path.

The path P is represented by a sequence of grid cells from the start to the target point. To enlarge the clearance of a way-point $p_i = (x_i, y_i)^T$ we need to find the obstacles perpendicular to the path segment through p_i. These obstacles can be found by ray-tracing orthogonally to the current gradient of the path segment.

Therefore we need to calculate the gradient for each way-point. We define the derivative of a

[2]We discuss the paper Stachniss and Burgard (2002) which is the most related work to our algorithm at the end of this chapter.



Solution
Obstacles
p_{i-1}
λ_i^1
p_i
p_{i+1}
ρ_i^1
$\nabla p_i = \frac{p_{i-1}-p_{i+1}}{2}$
$\lambda_i^1 = \binom{x_i}{y_i} + 1 \cdot \binom{\partial p_i/\partial y}{-\partial p_i/\partial x}$
$\rho_i^1 = \binom{x_i}{y_i} + 1 \cdot \binom{-\partial p_i/\partial y}{\partial p_i/\partial x}$

Figure 6.3: Derivation of solution

way-point by

$$\nabla p_i = \nabla (x_i, y_i)^T = \frac{p_{i-1} - p_{i+1}}{2}$$

The left and right environment around the way-point p_i can be found by

$$\begin{aligned} \lambda_i^{\varepsilon_l} &= \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \varepsilon_l \cdot \begin{pmatrix} \partial p_i/\partial y \\ -\partial p_i/\partial x \end{pmatrix} \\ \rho_i^{\varepsilon_r} &= \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \varepsilon_r \cdot \begin{pmatrix} -\partial p_i/\partial y \\ \partial p_i/\partial x \end{pmatrix} \end{aligned}$$

where ε denotes the step size, i.e. the number of grid cells searched around p_i. The derivatives $\partial p_i/\partial x$ and $\partial p_i/\partial y$ are the x and y coordinates to the left and the right of the way-point p_i. We apply λ^{ε_l} or ρ^{ε_r} and check if the grid cell $\lambda_i^{\varepsilon_l}$ or $\rho_i^{\varepsilon_r}$ are occupied. The new way-point is found by applying either λ or ρ $((\varepsilon_l + \varepsilon_r)/2)$-times depending on whether a collision was found on the left or on the right side of the way-point. The ε are bounded to a small number (we use a bound of 15 cells in our implementation) to search only in the neighborhood of p_i. Figure 6.3 shows an example.

In each cycle of the algorithm we check if there exists a path from a previous cycle and if this path is still valid for the current situation. This validity check comprises the coordinate transformation of the previous solution into the current occupancy map and it checks if the path collides with any obstacle in the world. If the path is still valid, we continue to use it, otherwise a re-planning is initiated.

Generating Motor Commands. Now we have calculated a safe path to the next target point. The robot will follow this path as long as no obstacles appear on it along the way. The collision-free path has to be realized, i.e. appropriate motor commands need to be generated. For the realization of the path we have to take the current orientation θ_r of the robot into account. If the robot is already moving it will not be able to perform, say, a 90° turn.

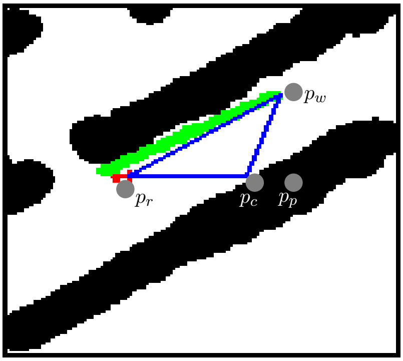


Figure 6.4: Collision-free triangle

Given the current orientation θ_r of the robot one can find the next obstacle in the direction θ_r by ray-tracing from the current position of the robot. The robot would collide with this point when, from the current point of time on, only translation is applied to the motors. Now we search for a point p_c on this line whose connection to the last "visible" point on the calculated path, denoted by p_w, is collision-free. Together with the robot position p_r we can define the triangle $\triangle(p_r, p_c, p_w)$. Each grid cell in this triangle can be guaranteed to be unoccupied by construction of the triangle. Figure 6.4 shows an example. The green line represents the planned path, the red dots represents the robot and its orientation, the blue lines yield the collision-free triangle. We construct this triangle to constrain the search for the motor commands to reach the target point in minimal time. For the calculations of the acceleration values below, we further need the following measures from the triangle: $d_w = |\overline{p_w p_c}|$, $d_c = |\overline{p_c p_r}|$, and $\alpha = \angle(\overline{p_r p_c}, \overline{p_r p_w})$. The point p_p is the orthogonal projection of p_w onto the line through p_r and p_c.

Inside this triangle we have to find appropriate translation and rotation velocity pairs which take us from p_r to p_w in minimal time.

To realize the path on the motor, i.e. to drive to the target point, we have to find a sequence of translational and rotational accelerations such that $S(t) = S(0) + V(t) \cdot t + \frac{1}{2} \cdot A(t) \cdot t^2 = p_t$ in minimal time. Here, $S(t)$ denotes the displacement, $V(t)$ the velocity, and $A(t)$ the acceleration of the robot at time t with the usual connections $S(t) = \int V(t)dt$ and $V(t) = \int A(t)dt$ between displacement, velocity, and acceleration for accelerated motion. We have to constrain the solution such that all points reachable with feasible $\langle a_{trans}, a_{rot}\rangle$ lie inside this triangle and thus are collision-free. $S(0)$ denotes the current location of the robot.

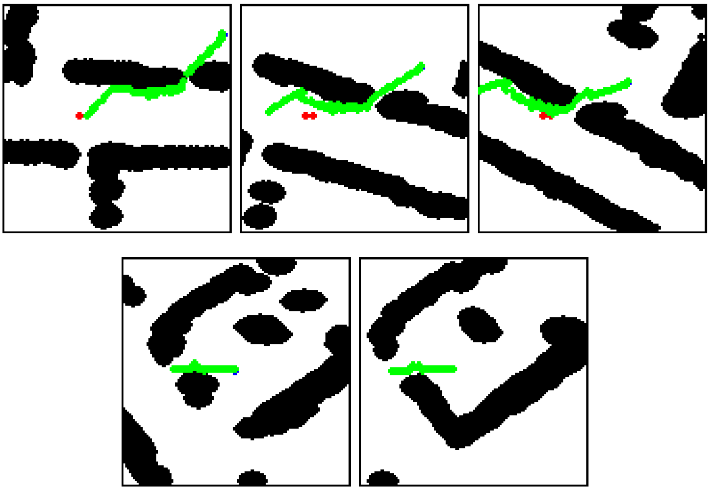
We approximate the problem iteratively by (1) discretizing the acceleration space (as described below) and (2) by approximating the displacement by $s_i = s_{i-1} + v_{i-1} \cdot \delta(t) + \frac{1}{2} \cdot a_{i-1} \cdot \delta(t)^2$ for each time step $\delta(t)$. The initial position of the robot is $s_0 = S(0)$, its initial velocity is $v_0 = V(0)$.

To find a sequence of motor commands taking the robot to the target position we apply A^* on the acceleration space of the robot. The current pose of the robot and the collision-free triangle yield the information we need to approximate the velocity values: we know that the robot has to turn by angle α to reach the target point and has to cover the distance d_w.

We search for rotational and translational accelerations in two separate steps. We do so to avoid the search in the five-dimensional pose-velocity space with a prohibitive branching factor for the search. The rotational velocities may take values $v_{rot} = c_{rot} \cdot a_{rot}^{\max} \cdot \delta(t) + v_0^{rot}$ with $c_{rot} \in \{-1, 0, 1\}$ during the search with $a_{rot}^{\max}$ the maximal rotational acceleration. We find a sequence of accelerations which cover the angle α in minimal time. It is obvious that when the angle α between the robot and the target position is less than $90°$ the time to reach the target is smaller when translation and rotation is performed at the same time than when the robot first turn towards the target point, stops, and then start to translate towards it. Note that if rotation minimizing the angle to the target point and translation towards to collision point is performed at the same time, all reachable positions lie inside the collision-free triangle. Driving along the edges of the triangle yields the worst solution to our approximation for which we can guarantee collision-freeness. Each path inside the triangle has a shorter path length and one needs shorter time to reach the target. If the $\alpha \geq 90°$ it takes less time first to turn towards the target, and then start to translate towards the target point. This consideration gives us a heuristic when to start translating towards the target point. If $\alpha \geq 90°$ the distance between the robot's position and the orthogonal projection of the target point onto the orientation line of the robot ($d_p = \overline{p_r p_p}$) is less than the distance between the robot's position and the collision point ($d_c = \overline{p_r p_c}$). This relates the search for translational velocities to the already calculated rotational velocities. This means that until we are oriented towards the target, our intermediate target distance is $d = \min(d_c, d_p)$. The objective of the A^* search for translational accelerations is to minimize the time to cover this distance d. The search branches over $v_{trans} = c_{trans} \cdot a_{trans}^{\max} \cdot \delta(t) + v_0^{trans}$ with $c_{trans} \in \{-1, -\frac{2}{3}, \ldots, 1\}$. $a_{trans}^{\max}$ is the maximal translational acceleration. For each node in the search tree we check if the required translational acceleration is admissible w.r.t. the given rotational acceleration given the kinematic constraints of the robot. Note that in each iteration the collision point p_c and the projection point p_p as well as the triangle changes. After one time step the robot's heading is towards a point between p_c and p_w. After several iterations the robot is oriented towards the target point and only translation is performed. Hence, we generate a sequence of velocities which takes the robot on a curve to the target point. The goal test for the A^* search is

$$\mathrm{IsGoal}(s_i, v_i) \equiv (s_i - s_\tau < v_i \cdot \delta(t)) \wedge (v_i - v_\tau < a_i \cdot \delta(t)).$$

where s_τ and v_τ are threshold values. With the $360°$ laser scanner the robot is also able to drive backwards to the target point. In the presented version of the algorithm we have a branching factor of 3 for the search for rotational velocities and a branching factor of 7 for the translational accelerations.


(b) Series 1

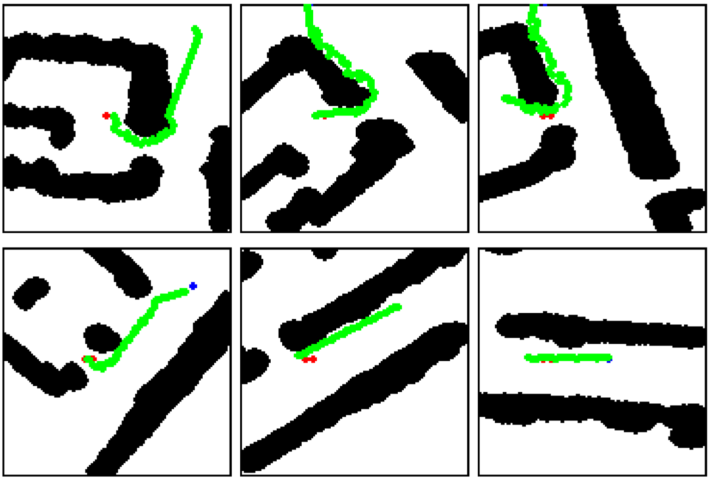

(d) Series 2

Figure 6.5: Example traces

Implementation Details

Special attention must be turned to an efficient implementation to make the navigation algorithm run fast on the robot platform. Main issues are the integration of the ellipses which are computationally costly, the path generation and the calculations of the motor commands in real-time. In the following we describe some details of our implementation and sketch some problems which arise on running the navigation algorithm on our platform.

Occupancy Grid. The step of integrating the sensor readings into the local map of the robot as ellipses is more costly than integrating single laser beams. Nevertheless, it is meaningful because collision detection of the robot with obstacles is eased. To efficiently integrate the obstacles, we pre-calculate the ellipses and assign a unique identifier to them. Thus, the cells an ellipse occupies in the grid can be taken from the library of ellipses and can be inserted at the appropriate coordinates in the grid. The size of the grid we use is $6 \times 6\ \mathrm{m}^2$ with a resolution of $5\ \mathrm{cm}^2$ which gives 14400 cells.

One problem with our soccer platform is that it is made of several levels which are connected with threaded bolts, blinding the laser range finder in four angle fields. To overcome these blind spots the robot builds a local map over several time steps.

Searching the Path. It is very important to calculate stable paths for two reasons: (1) if a path is reused in the next step one can reduce the amount of computational power needed; (2) oscillating behavior can be reduced.

A simple test before searching for a path to the target point reduces the run-time of the algorithm noticeably: we test if in the occupancy grid the target position is directly reachable for the robot. The test performs a ray-tracing from the robot position to the target position and the search is skipped if no collision occurs on this traced trajectory.

The implementation of the A^* search calculates a path of a length up to 300 grid cells (i.e. a path length of $6\ \mathrm{m}$) in less than $10\ \mathrm{ms}$ on the Pentium-III 933 machine on the robot. Given that the frequency of the laser range finder is $10\ \mathrm{Hz}$ and the navigation module runs at $20\ \mathrm{Hz}$ (not to lose any update from the laser range finder) there are about $40\ \mathrm{ms}$ left for the other steps of the algorithm.

We further reduce the computational costs with an extra step in the algorithm by displacing the path away from near obstacles. Especially in narrow door passages this extra step prevents searching for a path too often. A situation where this extra smoothing step pays off can be seen in the third picture of Figure 6.5(b). Here, the path is smoothed in such a way that the path does not need to be altered during the passage through the doorway.

Escape Situations. Another problem occurs when the robot navigates itself near walls such that its position in the grid lies inside an obstacle in the next step of the algorithm. For such rare

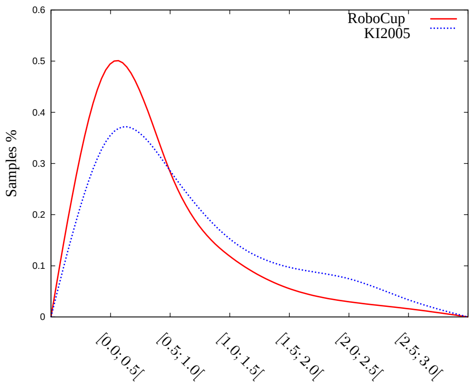


Figure 6.6: Velocity distributions of ROBOCUP Middle-Size league games

occasions an escape mode is implemented working on raw laser data. Upon these data the robot determines which direction the most unoccupied space exists and drives towards it very slowly.

Experimental Results

We use the proposed navigation and collision avoidance method for over five years with our soccer robots. Figure 6.5 shows the path visualization of two runs at our department hall. The red dot represents the robot, the green line represents the planned path, the black objects are the walls of the hall. In Figure 6.5(b) the robot should navigate from the hall into a room. One can observe that the path is calculated in such a way that the shortest possible connection between robot and target is chosen. The second picture in the first series shows the robot several snatches later. The chosen path is still valid, but one can notice that calculation of the drive commands and/or the realization of these commands on the robot are approximated as the robot does not stay on the path. But note that the position of the robot is still inside the collision-free triangle. In the fourth picture of Series 1 the robot entered the room; re-planning is initiated. Series 2 (Figure 6.5(d)) shows how the robot drives out of a room and along the hallway. In the second series one can note how the smoothing step influences the path. When driving out of the door the path snuggles around the left durn. Here, the path is diverted several grid cells away from the durn. This is the reason why in the second picture of series 2 the path remains the same. The snapshots for both series are taken from our simulation environment.

RoboCup Experience. In the Middle-Size league four to five fully autonomous robots in each team with a maximal size up to $50\ \mathrm{cm} \times 50\ \mathrm{cm} \times 80\ \mathrm{cm}$ ($\mathrm{w} \times \mathrm{d} \times \mathrm{h}$) are competing. During recent years we tested our navigation algorithm on many ROBOCUP tournaments. The requirements for navigation in the ROBOCUP domain are different from those in office environments. The robot's navigation algorithm must not be too cautious, slight contacts must be accepted, otherwise, the robot will never get the ball — the opponent robots are not polite at all. For the ROBOCUP setting we configure the navigator in such a way that the security distance is $0.0(!)$ and the obstacle ellipses are only extended to the half of their normal size. The data we logged on the ROBOCUP tournaments are the time stamp of the data, the localization position (which is very accurate, cf. Chapter 6.4), and several other data about the ball, the opponents, etc. Unfortunately, we do not log the motor velocities; moreover, the logger runs twice as slow as the navigator. So, for the evaluation here, we reconstructed the velocities from the log file by approximating $v = \Delta s / \Delta t$, taking only those values into account where $v > 0.1\ \mathrm{m/s}$.

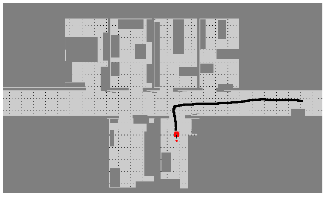
We analyzed the games from the World Cup 2004, the German Open 2004, the German Open 2005, and also four friendly matches. In total, the data were taken from 26 ROBOCUP matches. The data also contain the values of our goal keeper. The goalie usually does not travel longer distances, therefore its velocity values due to acceleration latency slightly drag down the values. In one friendly match at the German National Conference on AI (KI2005) we played on a handball field with a size $20\ \mathrm{m} \times 40\ \mathrm{m}$ (which is 8 times larger than an ordinary ROBOCUP field). One can observe slightly higher velocities in this game. Fig. 6.6 shows the velocity distribution over the velocity samples from the log files. We built 6 velocity categories, ranging from slow (between 0 and $0.5\ \mathrm{m/s}$) up to very fast (2.5 to $3.0\ \mathrm{m/s}$). The red curve shows the ROBOCUP distribution, the blue curve shows the distribution at KI2005. During these events the team traveled a total distance of $5.357\ \mathrm{km}$ (a soccer field has a size of $8\ \mathrm{m} \times 12\ \mathrm{m}$). The median speed lies around $0.69\ \mathrm{m/s}$. The maximal velocities lie around $2.9\ \mathrm{m/s}$. The Quartile of the distribution lies at $0.3\ \mathrm{m/sec}$, the 75-quantile has a value of $0.9\ \mathrm{m/s}$. These values were calculated using 98,666 velocity samples.

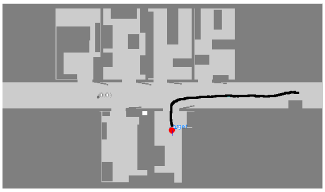
Note that in the Middle-Size league many maneuvers like approaching the ball are performed with medium to slow speeds. Nevertheless, we reach an acceptable median speed for this domain and severe collisions do not occur.

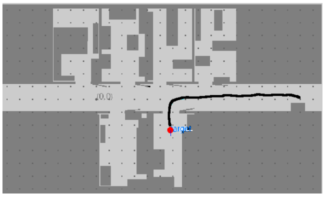
We also tested our navigation algorithm on our B21 robot Carl. From the performance point of view we encountered no problems with the algorithm. We could observe that Carl reached higher velocities than with the Dynamic Window (DW) approach (Fox et al. 1997) which is part of the original control software of Carl. The DW approach has inherently problems with narrow doorways as well as with relatively sharp turns.

Stachniss and Burgard (2002) report on similar results. They also compared their A^* collision avoidance method with the dynamic window approach. The A^* approach to collision avoidance clearly dominates the DW approach.

The next interesting question to be raised is how our algorithm with the heuristic to find rotation and translation velocities compares to their search method in terms of path length, traveled time, and average speed. As we cannot directly compare the methods, we have to compare it in a setting similar to the one they report on in (Stachniss and Burgard 2002). The scenario is nearly the same as in Figure 6.7(b). We get the same average velocities as Stachniss and Burgard report on


(a) Carl with DWA ($v_{\max} = 0.45$ cm/s)


(b) Carl with Navigator ($v_{\max} = 0.45$ cm/s)


(c) Carl with Navigator ($v_{\max} = 0.95$cm/s)

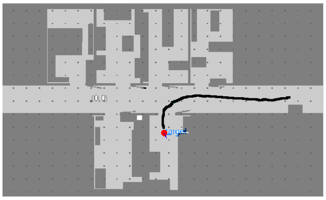

(d) Soccer robot with Navigator and ($v_{\max} = 3$ m/s)

Figure 6.7: Comparison of the DWA with our method

with a similar path length: the average speed for the depicted path was $32\ \mathrm{cm/s}$, the path length is $12.21\ \mathrm{m}$ with a travel time of $38.1\ \mathrm{s}$ with Carl's maximal speed set to $40\ \mathrm{cm/s}$ (vs. average speed: $34.3\ \mathrm{cm/s}$, path length: $11.86\ \mathrm{m}$, travel time $34.5\ \mathrm{s}$). The improvement compared with the DW approach is the same. Comparing Figure 6.7(a) and Figure 6.7(b) which show the trajectory of Carl driving with the DW approach and with our approach one can observe a smoother and faster trajectory. The improvement of the trajectory is comparable to (Stachniss and Burgard 2002). Fig 6.7(c) shows the trajectory of Carl in the same situation with a maximum speed of $0.95\ \mathrm{m/s}$. In comparison to Figure 6.7(b) there is almost no difference in the trajectory while the travel time to the target is cut in half. This shows that our navigation algorithm scales with increasing speed.

The trajectory in Figure 6.7(d) depicts the movement of one of our soccer robots with a maximal speed of $3\ \mathrm{m/s}$ in the same environment. The trajectory length differs only slightly from the other cases, the maximal speed was $2.85\ \mathrm{m/s}$, the robot had an average speed of $1.77\ \mathrm{m/s}$. It took $6.7\ \mathrm{s}$ to approach the target.

This shows that decoupling the search for rotational and translation accelerations in order to avoid a search over the five-dimensional pose-velocity space yields the same results in terms of path length, average speed, and travel time. Though, our method is much more reactive, is able to drive with much higher velocities, and consumes less computation time.

6.4 Laser-based Localization with Many Distance Measurements

Self-localization in dynamic environments is a central problem in mobile robotics and is well studied, leading to many satisfying approaches which can be found in the literature such as (Gutmann et al. 1999; Fox et al. 1999; Dellaert et al. 1999).

Most localization algorithms follow a probabilistic approach. The most popular among these is the Monte Carlo Localization algorithm (MCL) (Dellaert et al. 1999). Many applications of this method use a laser range finder (LRF) for perceiving the environment. MCL with LRF works best in environments with many landmarks.

In environments where landmarks are sparse, on the other hand, the results are far less satisfying. One such domain is the Middle-size league of robotic soccer, where up to ten mobile robots are competing on a field of the size of 12×8 meters. Available landmarks are the goals and the corner posts. To make the task even harder, they are often occluded by other robots.

For these reasons, the combination of a LRF and MCL is presently not the method of choice for self-localization in RoboCup. Most teams use vision-based systems for the purpose of localization. Nevertheless, our team, the “AllemaniACs”, successfully deploys a Monte Carlo approach with a laser range finder in that environment. In this section we present two modifications to MCL with LRF which allows it to be a viable and robust method for self-localization in RoboCup.

In principle, the sparsity of landmarks can be dealt with by taking many single measurements in a sweep from the laser scanner. However, this makes the use of the standard MCL algorithm intractable because the range of weights for the samples grows exponentially in the number of sin-

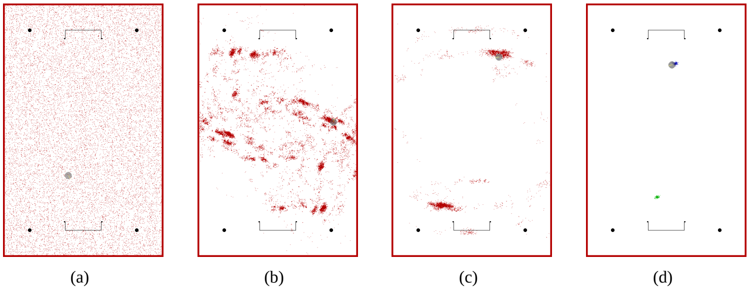


Figure 6.8: Simulated global localization on the RoboCup field. The real position of the robot is depicted by the gray circle. Because of the symmetry of the environment model, two clusters have developed.[4]

gle readings. To circumvent these computational problems we propose a heuristic weight function. Furthermore, we introduce so-called *don't-care* regions in maps that ignore the regions outside the field and thus enables incomplete specification of environments.

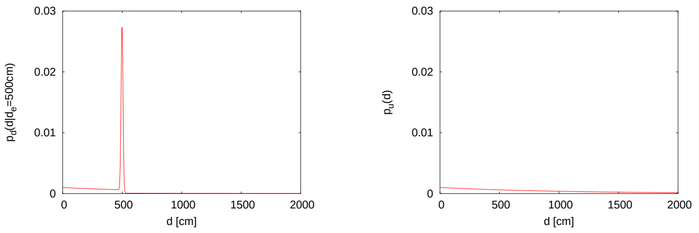
It turns out that our approach, which was inspired by the RoboCup setting, scales very well for indoor navigation in large environments.

A Heuristic Perception Model for MCL

In this section, we present the modifications to the MCL method to be able to localize with a laser range finder in environments with sparse landmarks. First, we briefly discuss *don't-care* regions and their integration into the sensor model. Then we introduce our heuristic weight function. The mathematical background of Monte Carlo localization can be found in Chapter 3.2.

Don't Care Regions in Occupancy Grid Maps. We use occupancy grid maps (as in (Fox et al. 1999)) for representing the environment. Each cell of the grid stores the probability of this cell being occupied by an obstacle. Figure 6.8 presents an example of a RoboCup field. Black regions denote an occupancy with a probability of 1 (the goals and the posts), and white regions are free areas. With the help of this information one can determine for each position on the map, which distance a laser ray pointing to a certain direction should measure. This value will be referred to as the *expected distance* in the following. The red border around the field in Figure 6.8 represent our *don't-care* extension to occupancy grid maps. In these areas simply no information about occupancy is given. This models an incompletely specified environment.

[4]This ambiguity cannot be resolved by MCL with LRF due to the inherent symmetry of the environment. In practice, it is resolved simply by using the information about the color of nearest goal provided by the camera used for ball tracking.



(a) Sensor model with an expected distance $d_e = 500\text{cm}$

(b) Sensor model for a don't care–region.

Figure 6.9: Sensor model for a single distance measurement for known and unknown expected distance.

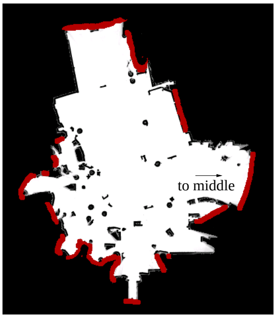
When using a 2D laser range finder, the weighting step of MCL can be performed as described in (Fox et al. 1999). The perception model $p(o^{(i)}|x)$ for a single laser beam i describes the probability that the laser beam traveling in a certain direction from x will measure the distance o. We need the environment model to distinguish two cases: (1) the laser beam will hit an obstacle; and (2) the laser beam will hit a don't care-region.

In the first case we know the expected distance of the measurement. According to (Fox et al. 1999), this yields the single-beam perception model shown in Figure 6.9(a). The peak represents a Gaussian curve assigning a high probability to the laser beam being reflected at the expected distance. Note that the probability of the measurement being shorter than expected is significantly higher than the probability of its being longer. This is due to the possibility of dynamic occlusion. Because the probability of occlusion is equal at all distances, it is modeled by a geometric distribution. The merging of the geometric distribution with the Gaussian yields the model displayed in Figure 6.9(a).

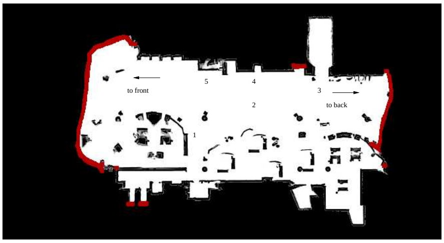
As one does not have information of an expected distance in the case when a laser ray is hitting a *don't-care* region the perception model is reduced by the Gaussian part. This yields a purely geometric distribution shown in Figure 6.9(b).

An example of a large map consisting of several parts are shown in Figure 6.10. The don't care regions define the stitch regions for the parts of the map. When the robot reaches such a red marked border of a map it internally switches to the next map. This example stems from a service robotic application we presented during an robotic exhibition in a large local bank. The robot navigated safely through the bank over several days.

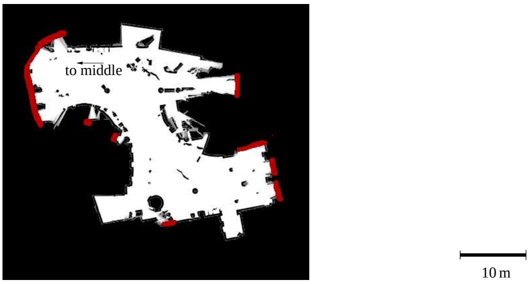
The Heuristic Perception Model. In order to perform the actual weighting of samples one has to combine the probabilities of single-beam perception for a given sample position x and a full 2D-sweep $o = (o^{(1)}, \ldots, o^{(n)})$ of the laser range finder by multiplying the weights (Fox et al.



(a) The front side.



(b) The middle. The numbers denote way points of the test run for robustness of the position tracking.



(c) The back side.

Figure 6.10: The building of a local bank which is about 80 m long. We used three intersecting maps to build an environment model of the whole floor. The *don't-care* regions, that are marked red, show the borders of each part of the map.

1999):

$$p_{mult}(z|l) = \prod_{i=1}^{n} p(o^{(i)}|x). \tag{6.1}$$

The weight range is exponential in the number of single measurements. With 360 readings it is practically impossible to weight the samples in that exponential range. To give an example, suppose that we have two positions x_1 and x_2 with uniform weights of 0.01 for x_1 and 0.025 for x_2. p_{mult} yields the following weights:

$$p_{mult}(x_1|o) = 0.01^{360} = 10^{-720} \quad \text{and} \quad p_{mult}(x_2|o) = 0.025^{360} \approx 10^{-576}.$$

The re-sampling step of MCL draws samples proportionally to their weights. This means that in this example the sample containing position x_2 has to be drawn 10^{144} times (!) as often as the sample with position x_1. Practically, this is impossible because one cannot handle a sample set as large as this.

Note that using the logarithm of p_{mult} does not provide a solution to this problem. One can handle the weight range by doing so and sample from the logarithm of the sample distribution. However, the resulting sample distribution would have to represent the proportions of the weights *before* using the logarithm. Thus, the sizes of the sample sets would still be intractable.

It may seem that the problem could be fixed simply by reducing the number of measurements to a manageable size.[1] However, in the RoboCup scenario this does not work since typically up to 90% of the readings are useless due to the sparsity of landmarks and occlusions. Hence even dropping a few readings risks losing the few precious good readings. Instead, we propose to use all readings for re-sampling, but replace the product by the sum of the measurements:

$$p_{add}(o|x) = \sum_{i=1}^{n} p(o^{(i)}|x).$$

In contrast to the multiplicative model the weight range is now linear in the number of single measurements of a 2D-sweep. Considering Figure 6.9(a) the weights range from about 0 to about $360 \cdot 0.025 = 9$.

Let us consider how the heuristics p_{add} differs from the mathematically correct model p_{mult}. First, it is easy to see that the former changes the proportions of the weights:

$$\frac{p_{add}(o|x)}{p_{add}(o'|x)} \neq \frac{p_{mult}(o|x)}{p_{mult}(o'|x)} \quad \text{for } \textit{most } o, o', x.$$

Furthermore, it does not preserve the order:

$$p_{add}(o|x) > p_{add}(o'|x) \not\equiv p_{mult}(o|x) > p_{mult}(o'|x) \quad \text{for } \textit{many } o, o', x.$$

[1]Experience shows that a reasonable number is in the order of 40.

Thus, the additive perception model may prefer positions to others that would not have been favored by the multiplicative model. A simple example for such a situation is the following: Let $o = (o^{(1)}, o^{(2)})$ and x_1, x_2 such that $p(o^{(1)}|x_1) = 0.06$, $p(o^{(2)}|x_1) = 0.01$, $p(o^{(1)}|x_2) = 0.04$, and $p(o^{(2)}|x_2) = 0.02$. Now it follows that $p_{mult}(o|x_1) = 0.0006 < 0.0008 = p_{mult}(o|x_2)$ and $p_{add}(o|x_1) = 0.07 > 0.06 = p_{add}(o|x_2)$.

What one can learn from this example is that the additive model tends to assign higher *overall* weights to positions with high *single* weights than the multiplicative model. Expressed in different terms, the low weights do not have such a great impact on the sample weighting as with the multiplicative model.

Having a look at Figure 6.9(a), low single-beam weights can have two reasons: (1) The reading is dynamically occluded, or (2) the reading is longer than expected. In the first case it is desirable that a low weight does not pull down the weight of a correct hypothesis. In the second case, however, the weighting function should reduce the weight of a position by the low single-beam weight. p_{add} works well in the first case while it fails in the second. In practice, however, it turned out that even in the latter case the correct hypotheses were supported and the algorithm converges.

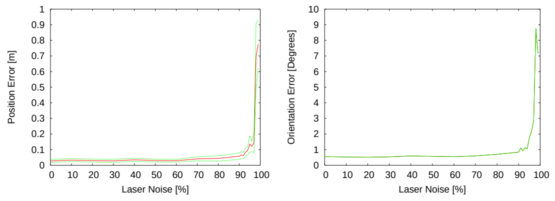
Experimental Results

We have tested our method extensively, both in simulation and with real robots at RoboCup events. We will now present results concerning the accuracy and robustness of our approach.

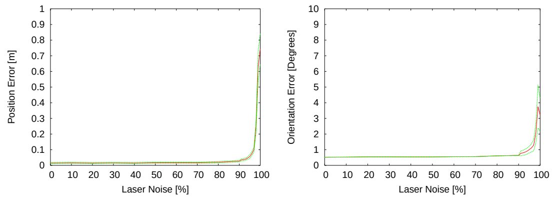
We used a map of a RoboCup field as shown in Figure 6.8 for the evaluation. It contains just the goals and the corner posts of a RoboCup field. We changed the noise level of the LRF in order to gain meaningful results. A noise level of $n\,\%$ means that we set $n\,\%$ of the single readings randomly shorter than the reading from the simulator. This simulates dynamic objects causing too short readings. The distribution for the random shortening was uniform.

During the experiment with movement the robot traveled at an average speed of $1.74\ \mathrm{m/s}$ and $34°/\mathrm{s}$, with a maximum speed of $3\ \mathrm{m/s}$, and $225°/\mathrm{s}$. Figure 6.11 shows that below a critical noise level of $90\ \%$ the accuracy is about $15\ \mathrm{cm}$ in the pose and $4°$ in orientation. Above a noise level of $95\ \%$ localization is no longer possible. One can get an intuitive understanding of what a loss of $90\ \%$ of the laser information means by calculating how many laser beams are still useful in that case. For example, if the robot is placed in the middle of the field, about $12\ \%$ of its laser measurements correspond to usable landmarks. A loss of $90\ \%$ means that only $1\ \%$–$2\ \%$ remain which means 3–7 distance measurements.

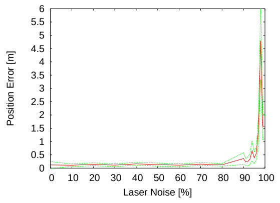
We gathered data from two RoboCup events the AllemaniACs took part in in order to gain results about the robustness of our localization approach. In order to do so, we counted dislocalizations during each of the matches. The results are shown in Table 6.1. All position losses were caused by severe failures of odometry due to slippage caused by collisions. When this happens, the robot senses a movement suggesting that it translated or rotated much farer than it actually did.



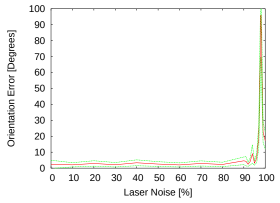
(a) No movement, simple map.



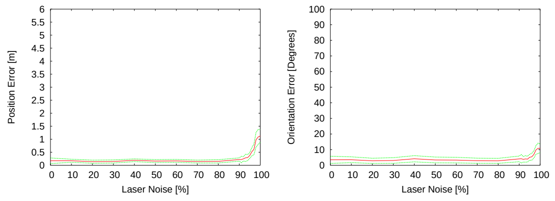
(b) No movement, complete map.



(c) Pose Error.



(d) Orientation Error.



(e) Fast movement, complete map.

Figure 6.11: Accuracy of position tracking by the MCL module. The red line shows the mean error, and the green lines represent a one–σ–environment around the mean.

RoboCup 2003							
Team	Artisti	Trackies	Eigen	Cops	Persia	ISePorto	AIS
Position Losses	0	1	7	4	2	1	5

German Open 2004						
Team	Paderkicker	Minho	Philips	FUFighters	Persia	AIS
Position Losses	0	0	0	0	1	1

Table 6.1: Position Losses

6.5 Fusing Sensor Values

Estimating the ball position accurately and reliably is one of the central problems in robotic soccer (RoboCup, (RoboCup 2006)), especially in the Middle-size league, where the robots often occlude the ball due to their size. Knowing where the ball is located is central for the cooperation and coordination task of the team. As the rules of the Middle-Size league allow for communication between the robots or to an external host, global sensor fusion can be applied for calculating a global ball estimate. Merging the single estimates of the robots into a global ball position is one of the keys to robust team-play as the perception of a single robot might be wrong. With the knowledge of the team-mates the wrong estimate of a single robot can be adjusted. This leads to a better overall team performance.

In this section we evaluate state-of-the-art sensor fusion techniques for merging the global ball position from the robot's local perceptions, filtering out wrong estimates and false positives. The methods comprise simple approaches based on averaging and more sophisticated ones, like the Kalman filter or the Monte Carlo approach (cf. Chapter 3.2). We tested the global sensor fusion with our team AllemaniACs during the World Cup 2003 in Padova, Italy, and 2004 in Lisbon, Portugal, and the German Open 2004 in Paderborn observing a significant increase in the quality of the ball position estimate. Moreover, we use the global ball information to detect when a robot is dis-localized.

We show that the best methods yield better results than reported before in the literature for the RoboCup setting and that the one reported in (Dietl et al. 2001) is outperformed by an even simpler one. The presented experiments were conducted in real game situations showing a significant increase in the availability and quality of ball position estimates.

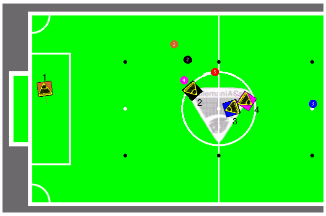
Applied Sensor Fusion Techniques

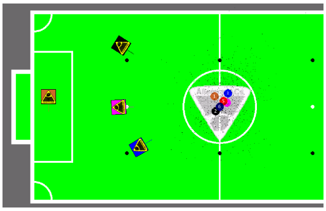
In this section we give a brief overview of the sensor fusion techniques we use in this case study. We start with simple ones like mean methods and go over to more sophisticated ones, like the Weight grid method, the Kalman filter and the Monte Carlo method. Finally, we test a combination between the Weight grid and the Kalman filter. This method is similar to the one proposed in (Dietl et al. 2001).

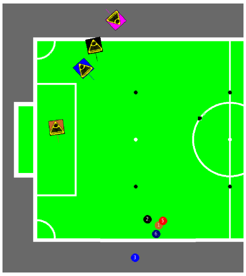
Arithmetic Mean. Each robot i seeing the ball contributes his local estimate $(lb_x, lb_y)^{(i)}$ about the ball position by communicating it to a central server. The global ball estimate (gb_x, gb_y) is calculated by averaging over the estimates from each robot, i.e. $gb_x = \frac{1}{n}\sum_{i=0}^{n} lb_x^{(i)}$, $gb_y = \frac{1}{n}\sum_{i=0}^{n} lb_y^{(i)}$.

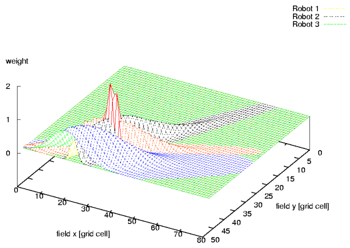
Here, every local ball estimate has the same importance. The estimate of a robot which is far away from the ball should not be weighted as much as the estimate of a robot being close to the ball. Therefore in a variant, we weight the estimates according to the distance from the robot to the ball and a time factor, which denotes how long ago the robot has seen the ball for the last time:

$$w_i = 1/dist_i \cdot conf_b{}^{(i)} \cdot conf_p{}^{(i)}. \tag{6.2}$$


(a) Arithmetic mean


(b) Monte Carlo ball localization


(c) Example situation for the weight grid fusion



(d) The weight grid for the situation in Fig. 6.12(c)

Figure 6.12: Fusion techniques

$conf_b$ is the ball confidence provided by the vision system. The role of this confidence is to give some means of persistence to the ball estimate. If the ball is not seen in one frame it is not reasonable to assume that the ball disappeared from the previously detected position as the vision system has some detection error. The confidence is modeled by a rapidly decreasing function over the time, which is set to 1 if the ball is detected. This confidence helps stabilizing the local ball estimates. The other confidence $conf_p$ is provided by the localization system and represents the confidence that the robot is located at the given position. The weighted mean is then calculated as $gb_x = \frac{1}{\sum_{i=0}^{n} w_i} \sum_{i=0}^{n} w_i \cdot {lb_x}^{(i)}$ and $gb_y = \frac{1}{\sum_{i=0}^{n} w_i} \sum_{i=0}^{n} w_i \cdot {lb_y}^{(i)}$. An example of the weighted mean estimation is depicted in Figure 6.12(a). The local ball estimates are enumerated and marked with the respective robot's color. The merged estimate is numbered as estimate 5 (red ball).

Weight Grid. This method uses a grid to represent positions on the field. The representation is similar to an occupancy grid representation (Moravec and Elfes 1985), but each cell can take weights greater than 1. For each ball estimate a 2-dimensional Gaussian distribution is calculated. The parameters for this Gaussian are taken from the *distance error* and the *pan error* which are provided by the vision system.

The cell update works as follows: each local ball estimate is weighted by the weight function given in Eq. 6.2. These weights are then multiplied with the Gaussian distribution, and the result is stored in the respective grid cells. Figure 6.12(c) shows an example situation where the ball can be seen by the goal keeper, the blue, and the black robot. The perception of the goal keeper (brown) is closest to the real ball position (dark blue) and its distribution is due to the weighting more narrow than ones from the other robots. The merged ball position is depicted in red. Figure 6.12(d) shows the weight grid representation of the scene.

Kalman. A Kalman filter (Kalman 1960) is used to estimate the state of a process variable $x \in \mathbb{R}^n$ in a dynamic system (see also Chapter 3.2). As we want to estimate the position of the ball we have $x = (gb_x, gb_y)^T$. The basic equation describing the stochastic process is $x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$ with measurements $z_k = Hx_k + v_k$, where w_k and v_k represent the process and measurement noise, resp. They are assumed to be independent of each other and normally distributed, i.e. $p(w) \sim N(0, Q)$ and $p(v) \sim N(0, P)$.

The matrix A represents the motion model relating the old process state with the new one. In our case we do not integrate a motion model directly as the local ball measurements from all robots are from the same point in time. Instead we propagate the global ball position by the ball velocity which is calculated using the last global ball estimates each time the Kalman filter is called. This simplifies the application of the motion model. Otherwise a complex motion model has to be found and applied. It turned out that applying the motion model this way works fine for the ball tracking task. To integrate control inputs in the estimate the variable $B \cdot u$ is added to x_k. In our case $u = 0$. Therefore, the state equation results in $\begin{pmatrix} gb_{x,k} \\ gb_{y,k} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} gb_{x,k-1} \\ gb_{y,k-1} \end{pmatrix} + w_{k-1}$

To integrate the sensor measurements z_k the matrix H is used denoting the observable components of x. In our case it is also the identity matrix as we can observe both coordinates of the ball. The process noise covariance matrix Q was empirically found as $Q = diag(3, 3)^T$, the measurement noise covariance matrix P is initially set to $P = diag(4, 4)^T$. For the first time when the filter is started we take the very first measurement from one robot as initial value for x_0. Applying the time update and measurements update equations (see. e.g. (Maybeck 1979) or Chapter 3.2) we estimate the global ball position from the robots' local estimates.

As another variant we use the *Kalman reset* filter. Here, the matrix P is reset to its initial value each time a global ball position is calculated, which in our case happens 10 times a second. This means that the actual measurement receives more attention.

Monte Carlo Localization. The *Monte Carlo (Ball) Localization* (cf. also Chapter 3.2 and Section 6.4) represents the distribution $Bel(l)$ with a set of weighted samples instead of the grid, which is used in the Markov localization. For the ball fusion we take samples that represent the ball position hypotheses. Therefore, a *sample* x_i consists of a possible ball position $l = \langle x, y \rangle$ and a weight w_i which is also called *importance factor*: $x_i = \langle \langle x, y \rangle, w_i \rangle$.

In one iteration of the Monte Carlo algorithm m samples from the sample set are drawn by chance according to their importance factor. A sample with a high weight is drawn more often than one with a lower weight. Additionally, some new samples are generated around new local ball estimates. Then, the algorithm works in two steps. First the motion model is applied to the drawn samples. This is done by promoting the samples according to the ball velocity and some noise. In the second step the samples are re-weighted with the new local ball estimates using the measurement error covariance matrix around the estimated ball position.

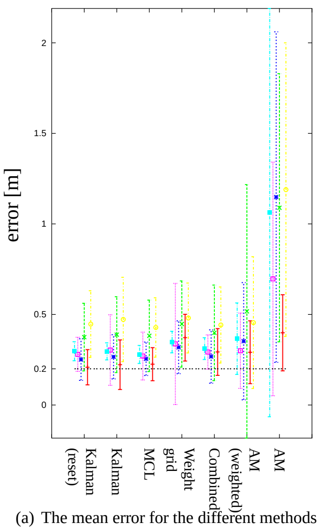
In practice it turned out that Monte Carlo is applicable with a set size of 1000 samples in our time constraints on our hardware. With more than 1500 samples we are not able to fulfill our time constraints of 100 ms of computation time any more. In Figure 6.12(b) the samples of the distribution approximation are depicted. The dark blue ball resembles the true position of the ball, the red one is the fusion result. The black dots show the samples.

Combined Fusion. Inspired by the work of the CS Freiburg RoboCup Team (Dietl et al. 2001), we combined the weight grid technique with the Kalman filter. All local ball estimates are used to calculate a *reference ball* by using our weight grid algorithm. We use only those local balls as input for the Kalman filter that are in between a radius of 1 meter around the reference ball. If the resulting ball of the Kalman filter is too far away from the calculated reference ball the Kalman filter receives a reset.

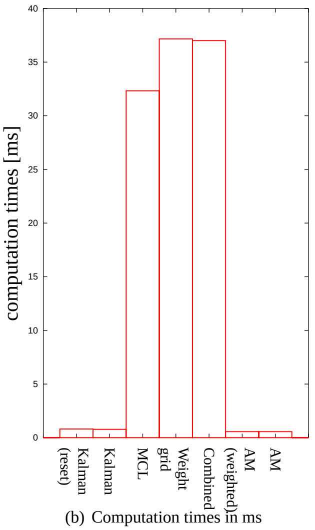
Experiments and Results

To get significant and realistic results about the quality of the presented fusion techniques we tested them in real game situations. We made several test games at the Philips RoboCup team in Eindhoven. To get ground truth for the real ball position, we used a ceiling camera. All relevant data were logged and can be replayed. The ground truth data were manually added to the log data. As this is rather elaborate, we do not have any ground truth data from the German Open or the RoboCup championships.

In the game against Philips, we picked 6 different sequences lasting between one and three minutes with significant action on the field. For example, in one sequence the ball was blocked between two robots so that the other robots of the team did not see the ball at all. One sequence is



(a) The mean error for the different methods



(b) Computation times in ms

Figure 6.13: Comparison of the methods

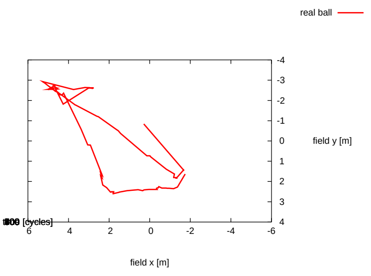
a typical game start situation, in another one the Philips robot kicked very often (and very hard, as usual) resulting in a rapidly moving ball. During some of the sequences one or more robots were dis-localized reporting wrong ball estimates.

The evaluation results are shown in Figure 6.13(a). The data represent the mean error in meters together with its deviations over each test run.[2] Not very surprisingly, the arithmetic-mean method yields the worst results, followed by the weighted arithmetic mean. The reason is that one outlier is enough to drag the merged position into the wrong direction. Even with weighting the estimates according to their distance to the ball this bias cannot be prevented. The grid-based method (combined and weight grid) are in the midfield w.r.t. their accuracy.

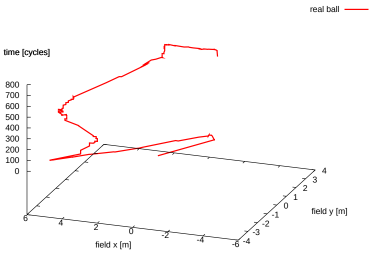
The first three places are taken by the Kalman, Kalman with reset, and Monte Carlo. Depending on the game sequence one of the three scored first place. One can see that in the mean these method have an error of about 20 cm. This is better than the results reported on in (Dietl et al. 2001). They reported on a position error of about 38 cm. Moreover, they conducted their experiments in a static setting, where the robots did not move during the experiments.

With respect to the computation times, one can state that the Kalman filter methods clearly outperform the other methods. With an average computation time of 0.8 ms and an error of about 20 cm the Kalman filter methods are accurate and fast. What surprised us most was that combining

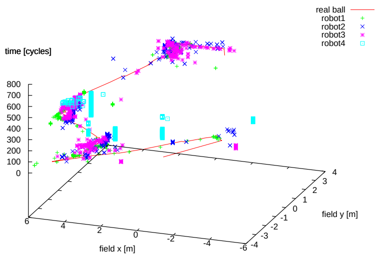
[2]For the whole testing time we had 8.520 cycles where the fusion module calculated a global ball estimate.



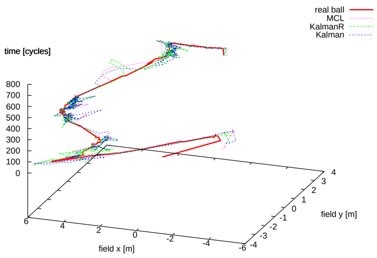
(a) Trajectory of the ball



(b) Trajectory of the ball over the time



(c) Sensor Data



(d) Tracking results of the Kalman and Monte Carlo methods

Figure 6.14: Sample Trajectories with their estimates

a Kalman filter with a weighted grid which is similar to the method proposed in (Dietl et al. 2001) does not seem to pay off, as a simple Kalman performed better, both in accuracy and computation time. To give an impression about the tracking of these methods we show one sample trajectory in Figure 6.14(a). Figure 6.14(b) shows the same trajectory over the time from a reverse angle. The tracking results of the two Kalman methods and the Monte Carlo method are depicted in Figure 6.14(d). To get a feeling about the quality of the tracking methods we moreover depicted the robots' local ball estimates in Fig 6.14(c).

In the next experiment we tested another typical RoboCup scenario: it happens very often in RoboCup that the referee picks up the ball, for instance after the ball passed the side line or was stuck between robots. Then, no robot can see the ball. The referee places the ball at some restart spot again. In this situation it is important to get stable ball estimates as soon as possible. In our second experiment, the robots took their kick-off positions and two helpers staying at the opposing restart points randomly placed a ball at one of these points.

In this experiment, again the Kalman reset method showed the best results. The Kalman reset had a mean error of 0.45 m, which is reasonable with respect to a minimum distance of 5 meters between the robots and the ball. Moreover, the computation time of 0.6 ms is a good result, especially compared to Monte Carlo which takes in the order of two magnitudes longer.

In the RoboCup domain it is very important to have a good estimate about the ball position. As the ball is perceived by only some robots most of the time, a stable global ball estimates helps to increase the team performance. The result for RoboCup's Middle-size league is that the Kalman reset filter shows the best performance. The quality gain using a ball fusion technique is significant. The availability of a global ball estimate lies at over 80 % in our experiments while each robot itself has seen the ball only 50 % of the time. Another nice effect of using a ball fusion technique is that one is able to detect when a robot is dis-localized. Comparing local and global estimates it is easy to notify when those values differ too much in order to detect a wrong localization position. Of course, it is crucial to find that value, as false positives must be avoided. It turned out that using the Kalman reset method we detected all dis-localizations during the test runs having only one false positive. Another problem is to gather ground truth data for empirical evaluation. In our experiments we installed a ceiling camera and associated the real ball position with the logged data by hand. This is a very time consuming process. Recently, Stulp et al. (Stulp et al. 2004) proposed a ceiling camera system to acquire ground truth data for the robots and the ball. With these data, we could evaluate the methods on a wider data basis. For the future we would like to conduct further experiments with real tournament data. As the presented results are very specific to the RoboCup domain, generally it turns out that one should try out several methods for the specific application domain in order to find the most appropriate method. Further, one can observe that applications of the Bayes filter (Kalman, Monte Carlo) provided the best estimates for the fusion task.

6.6 Robotic Soccer in the Middle-Size League

In Chapter 5 we showed an example of how READYLOG can be used to implement soccer agents in the Simulation league and in the interactive computer game UNREAL TOURNAMENT 2004. We showed how probabilistic programs can be used for deciding pass partners in a double pass scenario and how tasks can be modeled on the border between full DT planning and programming. In this section we will address the decision making applying decision-theoretic planning on a real robot in the soccer domain.

In the previous section we showed the low-level control software of our robots. These control modules are, like in the Simulation league connected to the high-level control by encapsulating them as basic actions and in form of a condensed world model. Basically in the Middle-size league, our robots can perform the following actions:

- $goto\_global(x, y, \theta)$: move to a global position on the field.

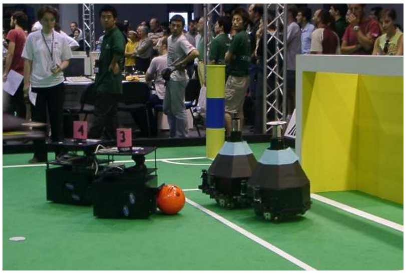


Figure 6.15: A scene from the RoboCup 2004 against the Osaka team (right-hand side).

- $goto\_relative(x, y, \theta)$: move relative x meter to the front, y meter to the side, and turn relatively with angle θ.
- $turn\_global(\theta)$ / $turn\_relative(\theta)$: turns the robot with angle θ globally or relatively, resp. One variant of this action is the $turn\_with\_ball$ action which tries the robot to turn without losing the ball in the gripper.
- $intercept\_ball(\theta)$: gain control over the ball such that the ball is in the gripper of the robot; θ is an optional argument which controls the angle the robot should have in the global coordinate system afterwards. As the robots use a differential drive, this action also implements that the robot depending on the intercept angle has to drive a curve around the ball.
- $dribble(x, y)$: the dribble action tries to reach a position on the field with the ball. This action takes the opponents into account and tries to find a path around all visible opponents and obstacles without losing the ball.
- $guard\_pos$: this is a defending action; the defender positions itself between the ball and the goal such that the body of the defender covers the goal.
- $search\_ball$: If the robot does not know where the ball is located, one can use this action to seek it. The robot starts turning slowly until the ball becomes visible again.
- $look\_at(x, y)$. This skill controls the pan/tilt unit of the robot. One can track a global world coordinate or the ball. The correct camera position is calculated from the sensor information about the position of the robot, the position of the object being tracked and the previous camera position.
- $kick(power)$: activates the kicker.
- $kick\_to(\theta, power)$ is a combined skill which turns the robot without losing the ball with θ and then kicks the ball.

- $move\_kick(x, y, \theta, power)$ dribbles the robot to the position (x, y, θ) in global coordinates and then kicks the ball.

The world information of the robot basically comprises fluents like the agent position, the position of the ball, and the opponents. All these information come together with confidences or visibility flags (which is true if the ball is seen), and come with two flavors: the agent could query the positions gathered locally from its sensors, or ask the information from the global world model. The information coming from the global world model are most of the times more accurate, though, they have some latency. The robot must store also tactical information. From the global world model the robot can retrieve the information about the roles of the robots like defender or attacker, or can directly address the robot which is, say, the defender. Having a role assignment for dividing up the tasks on the soccer pitch, one also needs a coordination mechanism to coordinate the behaviors of the different robots. The decision who should intercept the ball cannot easily be made by the robots themselves. The reason is that they often have only partial knowledge about the world, for example they sometimes cannot perceive the position of their teammates. Therefore, they could not decide which player might be located best to the ball to take an intercept action. The robots could negotiate about which robot is heading to the ball, but a lot of communication between the robot is then needed. Due to the time constraints this is not an option. An easier way is to synchronize this information through the global world model. This instance has all the information needed to decide which robot is positioned best. Based on a heuristics taking into account the distance and the angle of the robots the so-called $bestInterceptor$ predicate is computed and sent to each robot. It contains the number of the robot which is best positioned to the ball.

This basically makes up the world model the agent can use. In the following code examples also some other fluents or predicates are used, but their meaning becomes clear from their naming. The main loop of our soccer robots is the following:

```
**proc** main
  **forever do**
    **withPol**($\neg positionValid$, treatDislocalization)
    || **withPol**($illegalDefenseOrAttack$, moveOutPenaltyArea)
    || handlePlaymodes
  **endforever**
**endproc**
```

The agent program runs in an infinite loop. The program runs therefore forever. To terminate it, a special play mode $quit$ is caught in the procedure $handlePlaymodes$ which then terminates the agent program. Inside the loop, it is first checked with highest priority whether the robot is still localized. If the confidence coming from the localization module of the robot is below a threshold value, it is assumed that the robot has lost its position. Then one could either start an active localization, stop the robot and wait for human interaction, or try to regain the position on the field by using the information of the other robots. For the soccer scenario, active localization is not a good idea as the robot has to drive around for a while until its position hypotheses converge again. Thus, it could disturb the play, or shoot self-goals, for example. Regaining the position with the information from the other robots works in general. The procedure is to estimate the position using the global ball position (which is distributed from the world model) and the local ball perception if the ball is seen by the dis-localized robot. Nevertheless stopping for safety reasons is the best possibility. The robot cannot damage the field, or other robots. A human operator can then re-localize the robot. With $\mathit{illegalDefenseOrAttack}$ it is checked whether the robot is located in the opponent's or the own penalty area. The rules of the Middle-size league allow only one robot to be located in the penalty area, and only for 10 seconds. After 7 or 8 seconds the fluent becomes true initiating the robot to move out of the penalty area with the mentioned procedure. With lowest priority the procedure $\mathrm{handlePlaymodes}$ is called.

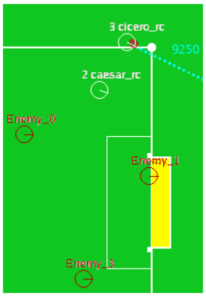
```
proc handlePlaymodes
  forever do
    exogfUpdate;
    if $\mathit{playMode} = \mathit{pm\_stop}$ then
      ( $(\neg \mathit{playMode} = \mathit{pm\_stop})?$ || procStop );
        elseif $\mathit{playMode} = \mathit{pm\_running}$ then
          (($\neg \mathit{playMode} = \mathit{pm\_running})?$ || procRunning);
          ...
      endif
    endforever
  endproc
```

In the Middle-size league a semi-automated referee system is used. The commands from the human referee are typed in by an assistant referee, and sent to the robots via wireless LAN. The different playmodes mentioned in the procedure $\mathrm{handlePlaymodes}$ are stop to immediately stop the game play ($\mathit{pm\_stop}$), a signal to go on with the game ($\mathit{pm\_running}$), and several other playmodes for indicating the kick-off, throw-ins, corner-kicks, and goal-kicks. The procedure $\mathrm{handlePlaymodes}$ now checks for the current playmode and calls other procedures which handle the currently set playmode. For each playmode a statement similar to those presented in the procedure $\mathrm{handlePlaymodes}$ is used. Each conditional which checks for a playmode stands in concurrency with other playmodes. For making the concurrent statement block in order to check for other playmodes, a test on the negation is used. If the condition of an if-statement holds, the then-branch is executed. Here, it is checked whether the opposite of the if-condition holds. This evaluates to false and thus the other branch of the concurrent statement is executed, i.e. the procedure which handles the playmode. As the semantics of the concurrent execution was to take a transition in either σ_1 or σ_2, and σ_1 fails unless the negated condition becomes true, the next transition of the procedure handling the actual playmode is executed. The next but one transition checks again if the negated condition holds. Summarizing, this construct guarantees that the appropriate procedure is executed as long as its condition holds. If the test action succeeds, i.e. the playmode must have changed, the else-branch of the conditional is further evaluated.

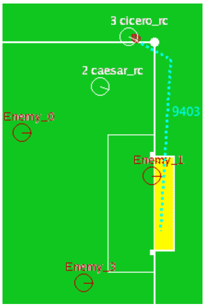
Next we want to show how the robot can make use of decision-theoretic planning. The procedure attackerBestInterceptor is a specialized procedure for the player taking the role *attacker* which is additionally the player which is best located to intercept the ball.

```
proc attackerBestInterceptor
  if $scoringSituation$ then scoreDirectly($own$)
  else
    if $\neg haveBall$ then interceptBall($own$, $fast$) endif
  endif
  solve(4, $reward$,
    continueSkill( $currentSkill$) ;
    ( $haveBall$? ;
    (kickTo($own$)
    | dribbleOrMoveKick($own$)
    | dribbleToPoints($own$)
    | if $isKickable$($own$) then
      pickBest( angle, $\{-3.1, -2.3, 2.3, 3.1\}$, /* in rad */
        (turnRelative($own$, $angle$, $medium$);
        (intercepBall($own$,$slow$ ); dribbleOrMoveKick($own$)
         | interceptBall($numberByRole$($supporter$);
           dribbleOrMoveKick($numberByRole$($supporter$)
        ) /* end pickBest */
      else
        interceptBall($own$); dribbleOrMoveKick($own$)
         | interceptBall($own$, 0.0)
      endif
    ) /* end solve */
  endproc
```

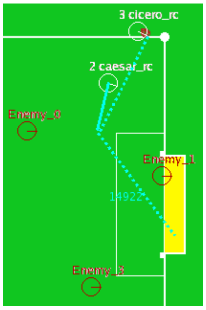
First it is checked reactively if there exists a situation where the robot can directly score by calling the procedure scoreDirectly. Otherwise, the robot tries to intercept the ball. With the **solve** statement the DT planning is initiated. To speed up the execution of the plan to be calculated the robot begins to calculate the policy before the intercept action succeeded. Therefore, the first action in the policy is to continue the previously executed action. In this case the robot will finish its intercept action. Then, as the intercept action may fail, we introduced another test if the robot has the ball in the gripper (which is part of the generated policy). This extra test is introduced to quickly detect if the policy is still valid when executing it (cf. Chapter 4.2.3 about execution monitoring of policies). Now, the agent checks for the different alternatives to find out the best one. It has the choices to try a goal shot (kickTo), to try dribbling followed by a goal kick whenever there is an opportunity to hit the goal (dribbleOrMoveKick), or to dribble to several tactical points on the field (dribbleToPoints). The latter action has lower reward, but it turned out that it is often



(a) Direct kick alternative.



(b) DribbleOrMoveKick alternative.

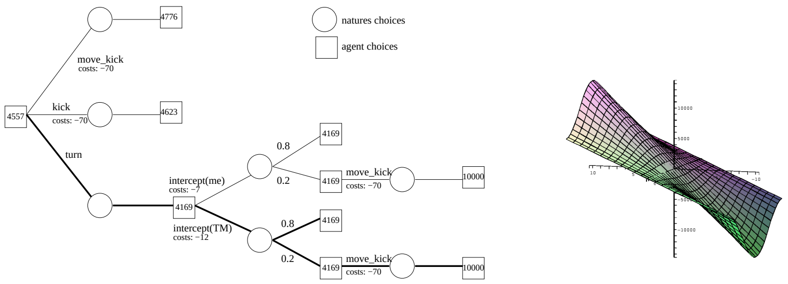


(c) Cooperative Play alternative.

Figure 6.16: Game situations

a good idea to dribble the ball towards the opponent goal, and then decide again what to do. The last alternative begins with a conditional which checks whether the ball is in the gripper of the robot. This conditional is introduced to further restrain the search space of the agent. In the case the ball is not kickable, the robot is going to try to regain control over the ball with executing an intercept action. Note that also the other alternatives described above are not possible if the robot is not in ball possession. As a last resort for the case that the first intercept action failed in execution, another intercept action will be initiated by the policy. Otherwise, if the robot has the ball, it makes use of the optimal choice of argument possibility, **pickBest**. Here, the robot calculates for the four given angles, if it should turn to the left or to the right. The idea behind this is that often the ball is blocked between the robots. With a turn to the side, the robot frees the ball again. Afterwards, it tries to intercept the ball and to dribble towards the opponent goal. Another alternative after the turn action the robot takes into account, is if the player with the role *supporter* might be able to take over the ball and dribble towards the opponent goal. As the actions are instantiated with the number of the acting robot (own, or $numberByRole(supporter)$) the execution system can distinguish if an action should be executed by the robot itself or if another robot is meant. In the latter case the execution of the action immediately returns with a success. By this simple mechanism multi-agent plans can easily be encoded (see also Chapter 5, p. 111).

Figure 6.16 shows three alternatives of the program $\mathrm{attackerBestInterceptor}$. Figure 6.16(a) shows the model of the $kick$ alternative. The robot receives a rather high reward for this action because the ball position in the projection is behind the opponent base line. It is though discounted because the ball is outside of the field in the projection. Figure 6.16(b) shows the action $\mathrm{MoveKick}$. The stochastic model of a move kick linearly approximates the curve the robot takes based on the target position and the initial angle of the robot. As one can see, the robot will not succeed with this action. Finally, Figure 6.16(c) shows the cooperative alternative, encoded by the **pickBest**



(a) Tree (b) Reward function

Figure 6.17: Plan tree

statement above. The robot Cicero chooses in this case a turn angle of $2.2\ \mathrm{rad}$. The model of the turn action is indicated by the dotted line in the figure. The robot taking the role *supporter*, Caesar in this example, now plans to intercept the ball and to kick it towards the opponent goal. This action receives the highest value of all alternatives. The rewards of the different alternatives are depicted in Figure 6.17(a). The evaluation is based on the reward function depicted in Figure 6.17(b).

The reward function mainly takes the position of the ball into account. The highest reward is awarded for the ball being in the opponent goal, and the lowest reward is given for the ball being in the own goal. Positions on the field are evaluated according to the reward function shown in Figure 6.17(b). Further it is checked if the ball is out of the field in the action models. In this cases the value is discounted. Besides the reward, a crucial point for applying the decision-theoretic planning approach are the stochastic action models. To give an impression how they are modeled, we show as an example the stochastic action of the intercept action below. It is very simplistic, but this elucidates the restrictions the robots have w.r.t. their abilities very well.

```
**procmodel** intercept(own, mode)
  $\exists angleToBall.angle(agentPos, ballPos, angleToBall? \wedge$
    $\exists pos.pos = interceptPose(angleToBall) \wedge pos = (x, y, \theta))$
  $sprob$((set($agentPos$($own$,$(x, y)$, $(0, 0)$));
    set(agentAngle($own$, $Angle$, 0)), 0.2, $isDribbleable$),
    ($nil$, 0.8, $\neg isDribbleable$),
    exogf_Update)
**endprocmodel**
```

In the preamble of the stochastic procedure in this case, we only use test actions. These have only a practical meaning, namely to bind the variable n. It is bound to a possible intercept position. First, the current angle between the agent's pose and the ball position is identified. Then with the auxiliary function $interceptPose$ the intersection between the robot's and the ball trajectory is found and set as the position where the robot will intercept the ball. With the $sprob$ statement, the nature's choices of this stochastic action are encoded. As $agentPos$ is a continuous fluent besides the calculated position (x, y) we also have to set the agent's velocity. It is assumed to be zero, as we only regard this situation as static. Besides the agent's position we have to set the estimated orientation of the robot at the intercept position. This is done with the $\text{set}(agentAngle)$ action. The probability of success for this outcome is set to 0.2 which is rather low. This resembles observations of the abilities of the robots to intercept the ball, and this low probability is due to the fact that, taking the dynamics of the game and the other robots into account, a lot can happen while the robot drives to the ball. The last condition $isDribblable$ checks if the intercept action was successful and is used to discriminate the different outcomes. Therefore, in the other case the negation of this is tested. The model for the failure case is even easier. As one cannot really predict how the world may evolve, we make no assumption about how the world might have evolved. Therefore the assumed world situation is the same as before. Why is this a good idea? Of course, one could define many different possible outcomes for the failure case. None of them will probably happen in reality. For the planning it does not make a real difference. For further planning the effect of this outcome is that the action has not happened at all, but during policy execution the sense condition $\neg isDribbleable$ is tested. This means, in the worst case, the robot will try to intercept the ball several times, as in the policy again an intercept action will be entered. Thus, the robot will eventually try to intercept the ball, and this is what the robot should do.

This gave an impression of how decision-theoretic planning can be used for soccer applications with real robots. It also showed that due to many limiting factors, the models and abilities for using the full range of expressiveness of READYLOG is restricted. This is also owed to the complexity of the planning approach. More complex models result in longer planning times, which for the robotic soccer case, are not feasible. The table below shows the computation times logged at the RoboCup World Championships 2004.

	examples	min	avg	max
without ball	698	< 0.01	0.094	0.450
with ball	117	0.170	0.536	2.110

In cases where the robot was not near the ball the computation times for the generating policies is rather low. In cases with ball, the agent has to take more possibilities into account and therefore has much longer computation times. But an average of 0.5 seconds are on the border-line of what is still possible for a soccer robot to reason about what to do before an opponent will steal the ball.

6.7 A Service Robotics Application

As another application for READYLOG we want to address a typical service robotics application and show an example from a RoboCup tournament. At RoboCup tournaments the teams have to participate besides the soccer matches also in a so-called Technical challenge , where the scientific progress is rated. There are several challenges to fulfill, for example one has to show that the robots are able to avoid obstacles on the soccer field. As part of this challenge, there exists also an

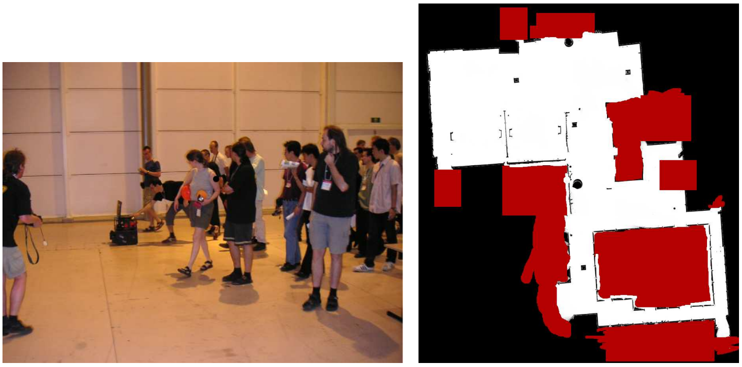

(a) The Robot driving through the exhibition hall. (b) Map of RoboCup Championships in Lisbon 2004

Figure 6.18: Technical Challenge at the RoboCup Championships 2004 in Lisbon.

open challenge, where teams can show whatever they like. At the RoboCup Championships 2004 we won the Silver Medal in the Technical Challenge by demonstrating a tour-guide application. The robot started at the so-called "team area" where all teams store their equipment and have the possibility to calibrate and program the robots. The referees were to choose one of the four soccer fields as destination for the tour. Then the robot calculated the shortest route to this field applying decision-theoretic planning. On the way several places of interests where announced by the robot.

Figure 6.18(a) shows our robot on the way through the exhibition hall. Figure 6.18(b) shows the occupancy map of the exhibition hall. On the upper half of the map one could see two of the fields, on the lower half there was the team area. We defined a topological map with "outstanding" sights like "grand stand" or "field one". The nodes of this map were made available to READYLOG by the fluent $mapNode$, and the relations $childrenOf(mapNode)$. The program the robot had was the procedure $\mathrm{pathPlan}$ given below.

```
proc pathPlan( $Goal$, $H$ )
  solve( $H$, $reward\_at(goal)$)
    while $\neg mapNode = goal$ do
      pickBest( $child$, $childrenOf(mapNode)$, gotoMapNode($child$) )
    endwhile
endproc
```

The action $\mathrm{gotoMapNode}$ is in fact a procedure which initiates the robot to drive to the respective coordinate and announce the exhibit. The reward function for the planning task was quite simple. At the goal node the robot receives a high positive reward and zero for all other nodes. When defining a metric on the graph and giving discounts for longer edges one easily could ensure that the robot will take the shortest path to the goal.

```
function $reward\_at(Goal)$
  $\exists v.((mapNode = Goal) \wedge v = 100$
    $\neg mapNode = Goal \wedge v = 0\ )$
return $v$
```

Several other such application like at a fair in a local bank demonstrated the robustness of our approach. At this application the robot also had to fulfill tour-guide tasks in a crowded bank for several days. The occupancy grid map we already showed in Figure 6.10 on page 162.

6.8 Discussion and Related Work

In this chapter we showed the application of READYLOG on a real robot system. Unfortunately, before one can run a high-level control language on a robot to plan something meaningful, one has to provide the whole robot system, hardware as well as software. Over the recent years of experience participating with a robot team at the Middle-size league, it turns out that special focus must be laid on a tight system integration. As stated in the introduction to this chapter, it does not help to have a good working high-level control when the low-level system is only fairly working, and vice-versa a good low-level system does not help to come to clever decisions without a good high-level control. This is the reason why we focused in this chapter on both, the low-level and the high-level control. Regarding our READYLOG applications it became obvious that good behaviors must be provided by the low-level system. Then, the language READYLOG can show its strengths. Decision-theoretic planning in READYLOG is well-suited for the decision making of a soccer robot. We also showed how a team of robots can be coordinated with multi-agent decision-theoretic plans in READYLOG. Nevertheless, as we also showed in Chapter 5.1 the drawbacks of the expressiveness of READYLOG are its planning times. With the hardware of our robots (Pentium-III 933) the planning times are at the border of what is possible for robotic soccer. The other application we showed, the service robotics application, is less time critical and therefore READYLOG is best-suited to solve the task. In the following we discuss the low-level modules of our robot system.

Collision Avoidance. Borenstein and Koren (1991) proposed to use vector field histograms. The target point exerts an attractive force to the robot while obstacles emerge repulsive forces. The trajectory of the robot is then formed by the sum of both these forces. They propose a special wall-following mode to avoid getting stuck in local minima which could otherwise cause oscillating behavior. The method was tested with robots equipped with sonar sensors driving with an average speed of 0.53 m/s. In (Fox et al. 1997), Fox et al. proposed the dynamic window approach. It is directly derived from the motion equations. In the velocity space circular collision-free trajectories are searched. To handle the state space they define a *dynamic window* around the robot's trajectory where only those velocities are regarded that the robot can reach in the next time interval. Finally, a trajectory is found by maximizing over the minimal target heading, maximal clearance around the robot and the maximal speed. The method was tested on an RWI B21 robot with a maximum speed of 0.95 m/s. A similar approach except for the dynamic window was proposed in (Simmons 1996). Recently, Dimarogonas and Kyrikopoulos (2005) proposed a collision avoidance scheme for independent non-holonomic non-point agents, other work, like (Ögren and Leonard 2005) proved the convergence of the Dynamic Window Approach, or, like in (Gottfried 2005), try to encounter the problem of collision avoidance by qualitative calculi, or like Pradalier *et al.* navigate with car-like robots in pedestrian areas (Pradalier et al. 2005). The most related researc h to our approach is the method proposed by Stachniss and Burgard (Stachniss and Burgard 2002). They use A^* to find an optimal trajectory to a given target. The state space used here is a five-dimensional pose-velocity space consisting of the pose x, y, θ and the translational and rotational velocities v, ω. First, a trajectory to the target is calculated using A^* in the $\langle x, y\rangle$-space. This trajectory deals as a heuristic for the search in the pose-velocity space. With a value iteration approach a 70 cm broad channel around the robot is calculated which determines the state space for the five dimensional search. Stachniss and Burgard tested their approach on a Pioneer I and an RWI B21 both having a maximum speed below $1\ \mathrm{m/s}$. In their experiments they describe that their approach is able to calculate a path of a length of $2\ \mathrm{m}$ in $0.25\ \mathrm{s}$ with the robot driving with a top speed of $0.40\ \mathrm{m/s}$ on a Pentium-III 800 MHz. They report that in the search for the velocity values they use a discretization of the grid of $10\ \mathrm{cm}^2$, they discretize the translational velocities in steps of $10\ \mathrm{cm/s}$ yielding 9 velocity steps and the rotational velocities with $\pi/16$ yielding 32 rotational velocity steps. This makes a maximal branching factor of 9×32 for the A^* search. Unfortunately, they do not report on the average branching factor which results from constraining the search space in their approach. The latter approach is especially interesting as it does not have problems with U-shaped objects (as opposed to (Borenstein and Koren 1991)) or narrow doorways (as opposed to (Fox et al. 1997)).

Localization. Bayes filtering using Kalman filters (KF) or Monte Carlo localization (MCL) approaches are standard methods for localizing a robot with proximity sensors. Implementations of the Bayes filter differ mainly in the representation of the belief. For example, Kalman Filter based approaches use a uni-modal Gaussian distribution (e.g. (Gutmann et al. 1999)). In contrast, grid-based Markov Localization (ML) stores the distribution in a discrete grid covering the state space. While the former methods are computationally more efficient and accurate, their restriction to uni-modal distributions makes it impossible for them to perform global localization (finding the position without initial knowledge). ML is able to solve this task, and also the kidnapped-robot problem, which means finding the correct position again after the filter converged to a wrong position. A combination of both methods, called ML-EKF (Gutmann 2002), combines the robustness of ML and the accuracy of the KF. MCL is a further refinement of ML, replacing the probability grid by the already described sampling mechanism. As the filter converges, the samples gather around positions of high probability. An experimental comparison (Gutmann and Fox 2002) of the described localization methods showed that the ML-EKF approach performs about equally well as MCL. Because MCL is quite popular, there are a number of extensions tackling shortcomings of the base algorithm. Here, we mention two of them. KLD sampling (Fox 2001) allows an adaptation of the size of the sample set in every iteration of the algorithm which greatly increases efficiency. Cluster-based sampling (Milstein et al. 2002) tackles the problem of symmetrical environments. In such domains, ordinary MCL will develop two or more clusters of samples and finally decide for one of them. This often happens prematurely. The clustering extension works around this problem by tracking emerging clusters within different sample sets until the sensory information allows for an unambiguous decision for one of them. Our aim was to localize our robots with a LRF on a soccer field. Here, the landmarks are sparse (two goals and four small corner posts), which means that the standard Monte Carlo approach could not be applied out of the box. We had to adapt the perception model for laser range measurements so that many readings can be used. While usual implementations use about 20–40 measurements per scan, we needed to make use of a whole sweep of 360° at a resolution of 1° because of the sparsity of landmarks and high sensor occlusion in RoboCup. Our heuristic perception model exhibited good performance in this setting. We regard the fact remarkable that MCL is at all able to keep track of the position in the presence of a constant laser noise of 90% and fast movement of the robot.

We ascribe the good performance of our MCL module in the presence of laser noise to the characteristic of the additive perception model (cf. Section 6.4) that it prefers position hypotheses with high individual weights. The drawback of the additive model is the weak effect weights have on the weighting of a position for readings that are longer than the expected distance. This plays a role primarily in global localization. However, we found out that our approach works well in this case, too, as is shown, for example, in Figure 6.8. Although the weights for wrong positions are not drawn down as strongly as with the multiplicative model, it turns out that the correct hypotheses still have a higher overall weight and are thus preferred. Concluding, the additive model is a heuristic which turned out to work very well in practice.

Sensor Fusion. The soccer domain is an interesting domain for research on sensor fusion. Most of the work concentrates on merging perceptions of the ball to one consistent estimate. The methods commonly used are probabilistic method as Kalman filters (Kalman 1960) or Markov Localization (Fox et al. 1999). As there exists a large body of work on sensor fusion (e.g (Brooks and Iyengar 1998; Chung et al. 2001; Hall 2004)) we here we will concentrate on the related work in the field of fusing ball estimates in the RoboCup domain. One can distinguish between the so-called *local sensor fusion* and *global sensor fusion*. In the former case the perceptions of several sensors on one robot are combined. The latter refers to merging the perceptions of different robots. Dietl et al. (2001) present a ball tracking algorithm, which combines a Kalman filter with Markov localization. They assign a new measurement to an existing track of observation by minimizing the sum of squared error distances. For predicting the ball position a Kalman filter is used. For this

Kalman filter they use Markov localization as an observation filter. They report a mean error of 38 cm for a moving ball while the robots did not move. Stroupe et al. (Stroupe et al. 2001) represent each ball estimate as a two-dimensional Gaussian in a canonical form. This allows to merge the single estimates of the robots simply by multiplying them. For predicting the ball position they use a Kalman filter approach. Pinheiro and Lima (2004) also represent sensor information about the ball as a Gaussian applying *Bayesian Sensor Fusion*. They assume that the last position is known and that the single estimates are close by each other. The team *Mostly Harmless* (Steinbauer et al. 2004) use local sensor fusion to integrate the perceptions from the different sensors their robots are equipped with. They use a Monte Carlo approach (Dellaert et al. 1999) to merge the data from the different sensors into a local world model. They also provide a merged global world model. The *Milan RoboCup Team* use an *anchoring approach* (Bonarini et al. 2001). The Sensor data are represented symbolically and are anchored with objects from the environment. For the sensor fusion of the symbolical sensor data they use fuzzy logics.

Chapter 7

Qualitative State Space Abstractions

7.1 Introduction

So far, we have seen several examples how READYLOG can be applied to real-world applications. While for the UNREAL TOURNAMENT 2004 domain the example we have chosen was a case study for the different specification possibilities of READYLOG w.r.t. high-level decision making, the set of fluents which we came up with for the soccer domain were kind of ad hoc. Besides common knowledge about soccer there is no compelling reason for exactly this domain specification. The question is if we can rely on more scientifically grounded approaches to specify our behaviors for the soccer domain. When looking at soccer literature the simple answer is that there are some theories about soccer which we can rely on. These are not theories in a mathematical sense but nevertheless the knowledge about soccer is encoded in such a way that we can use it. In the next section, we present our approach to come up with a theory for robotic soccer. For the specification, the language READYLOG turned out to be very expressive.

While working on formulating abstract soccer patterns in READYLOG we observed that most of the patterns used in the soccer literature are qualitative in nature. Humans usually use qualitative notions for positions on a soccer pitch, assuming common sense knowledge about its meaning. For example, all the soccer patterns in (Lucchesi 2001), the book which we used for our investigation, are diagrams with circles denoting qualitative positions on the pitch, and arrows denoting the actions of the players. To simplify the behavior specification for these soccer moves we constituted the most important qualitative predicates and found mathematical models for them. In Section 7.3 we discuss our approach in detail.

Having parts of behavior specification inspired by human soccer theory and a qualitative world model which helps to implement the qualitative notions of soccer, we applied both to Simulation League. In Section 7.4 we report on some of our results about establishing a decision-theoretic plan library for simulated soccer. The idea we follow is rather simple. Instead of calculating a particular decision-theoretic policy, we leave the whole policy abstract. In a sense, we store the whole DT computation tree. When the agent now faces a particular situation again, it just picks one of the already stored policies from the library and executes it. By this the agent can significantly reduce its on-line computation time. As a nice effect of the plan library it turns out that macro action similar to those proposed in Chapter 4 can easily be modeled with it. This has the additionally advantage that macro action here do not rely on explicit state enumeration any longer. We conclude this chapter with a summary and a discussion of more related work in the area of qualitative world modeling.

7.2 Formalizing Soccer Strategies

In this section we describe our approach to formalize soccer strategies for soccer robots. This work was done together with Dylla et al. (2005). The idea is to derive the basic behavior patterns of soccer as described in (Lucchesi 2001) and condition them to the different soccer leagues in RoboCup. The formalization of the soccer moves was done in GOLOG which coming with its formal semantics, is also very suited for this task.

In the original paper (Dylla et al. 2005), we also started a case study how to apply the soccer moves to the different leagues. We leave out this part and concentrate on the qualitative aspects of the formalization referring to (Dylla et al. 2005) for further details.

In the following we describe first the basic ontology for soccer strategies and related with this, the basic actions of a soccer player before we derive the basic qualitative predicates needed to formalize soccer moves. Further, we give an example specification of a soccer move in GOLOG which can nearly directly be encoded on a soccer robot as a READYLOG program.

7.2.1 The Organization of Soccer Knowledge

Among the modern soccer publications Lucchesi's book (Lucchesi 2001) is one of the most interesting because it concentrates on strategic aspects of soccer rather than on training lessons. Soccer strategies in the literature (e.g. (Lucchesi 2001; Peitersen and Bangsbo 2000)) are not as highly structured as for example strategies for American football are, though they are structured enough to build a top-level ontology for it. According to (Lucchesi 2001) there are two phases in the game: (1) the defensive phase and (2) the offensive phase. In the defensive phase the ultimate goal of the team is to *prevent the opponent to score a goal* and to *gain ball possession* again. When the second sub-goal of this phase is fulfilled the game enters the offensive phase. Here, a controlled *build-up of the play* has to be performed. In general, there are two ways to build up the play: either we introduce the phase in a counterattack manner, fast and direct with a long pass or deliberately by a diagonal pass or a deep pass followed by a back pass. The taxonomy for soccer strategy is depicted in Figure 7.1, where "3-4-1-2" stands for the basic tactical setup of the team.[1] In the following, we will concentrate on the building-up phase for an illustration of deriving basic behavior patterns for soccer play. Figure 7.2 shows two example diagrams from (Lucchesi 2001). The goal of the attacking team is now to *create a scoring opportunity*. The finalizing move is to try to *score a goal*. In soccer, there exist several strategic groups, each having a particular task in fulfilling the just mentioned strategy. The defense has to prevent the opponent team to score and has to build-up the play. The mid-fielders work to create a scoring opportunity and the offense has to score the goal.

[1]The pattern 3-4-1-2 means that the team is playing with three defenders, four midfielders, one offensive midfielder, and two forwards.

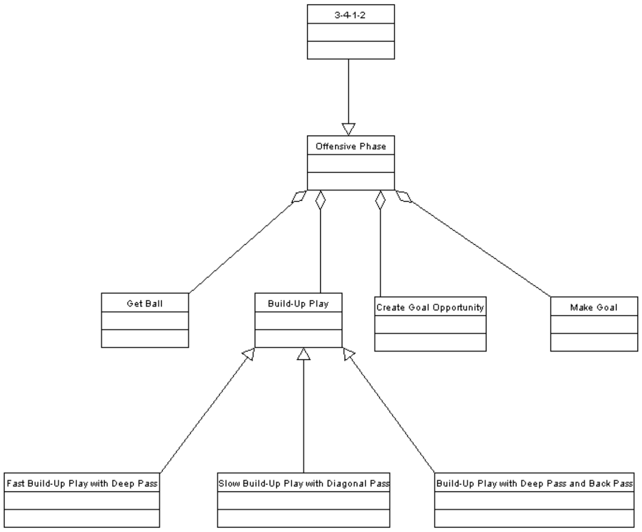

3-4-1-2
Offensive Phase
Get Ball
Build-Up Play
Create Goal Opportunity
Make Goal
Fast Build-Up Play with Deep Pass
Slow Build-Up Play with Diagonal Pass
Build-Up Play with Deep Pass and Back Pass

Figure 7.1: Top-level ontology according to (Lucchesi 2001).

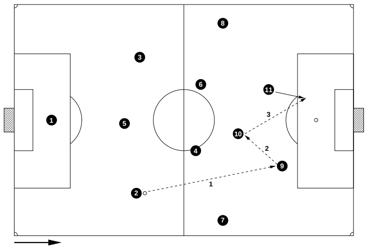
Accordingly a soccer strategy can be defined as a tuple

$$str = \langle RD, CBP \rangle.$$

With RD as a set of *role descriptions* that describe the overall required abilities of each player position in relation to CBP, the set of *complex behavior patterns* is associated with the strategy. Given the strategy str, the associated role description $rd \in RD$ can be described by the defense tactics task, the offense tactics task, the tactical abilities, and the physical skills.

An Example: Build up Play

In this phase of the game the team's objective is to take the ball towards the opponent's goal in order to establish a setting which allows for creating a scoring opportunity. There are a number of ways to build up a play: (1) *immediately with a long pass*. The long pass enables the team to take the ball up-field towards the opposing goal very quickly. There is an immediate reversal of play and the risk of losing the ball near ones own penalty area is very low. However, the long pass is difficult to receive, so the opponent may be able to steal the ball more easily. Moreover, as there is not much time, the team cannot move forward in a coordinated way; (2) *deliberately with a diagonal pass.* The diagonal pass allows for a coordinated way to move forward with the ball and it is easy to receive. The time to get close to the opponent's goal is longer than with the long pass. Thus, chances of losing the ball in a dangerous area are higher; (3) *deliberately with a*


(a) Diagram 4 from (Lucchesi 2001).

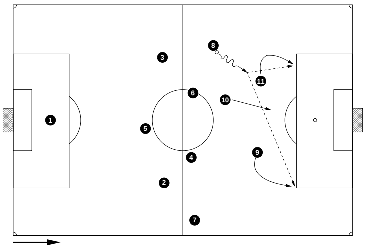

(b) Diagram 21 from (Lucchesi 2001).

Figure 7.2: Two tactical diagrams from (Lucchesi 2001). The bold arrow next to the field indicates direction of play. Player movements are represented by arrows (→ or ↷), passes are indicated by dashed arrows (⇢), and squiggly arrows (⇝) stand for dribbling. Opponents are not shown.

deep pass and a subsequent back pass. This way of building up play requires very good timing as it involves three players who have to move in a coordinated way. If such a move is carried out successfully it allows the team to move forward up-field without great risks if they lose the ball. We depict one exemplary tactical move for each of the three patterns to build up play mentioned above in Figure 7.3.

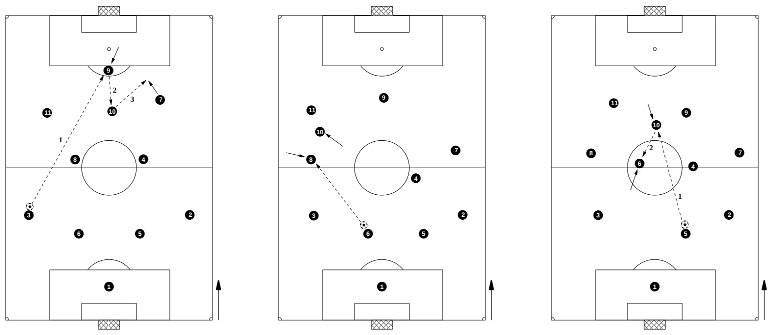
The decision which pattern to choose certainly depends on the opposing team as well as on the particular situation. That is to say, when playing against an opposing team which has many players in the midfield one would perhaps favor to build up a play with a long pass whereas with a team leaving lots of space uncovered in the midfield one would prefer the deep pass.

Basically, all the possibilities mentioned above are meant to take the ball up-field and establish a more offensive setting for the team while remaining in possession of the ball. The ball is either taken forward from the defense to the midfield section or in the case of the long pass directly to the offense section. Both can be done through the center of the playing field or by using the wings of the pitch. Depending on how the play was built up, there are several ways to create a scoring opportunity.

Basic Primitives

The previous examples already suggest which behavior patterns are needed for a soccer formalization. Following the lines of (Lucchesi 2001), we distinguish between *role* (back, midfield, forward) and *side* (left, center, right) in soccer. This distinction is more or less independent from the pattern of play (e.g. 3-4-1-2 or 4-2-3-1). The combination of role and side (e.g. center forward) can be interpreted as *type* of a (human or robotic) soccer player or as *position* (region or point) on the soccer field. Therefore, we basically have nine different positions, as illustrated in Figure 7.4(a).

The notions player type and position can be seen as instances or specializations of the notion of an *abstract position*, or address for short, usually associated with its (actual) coordinates or a region on the soccer field. Also the ball (strictly speaking, its position) is an abstract position,



(a) A long pass (b) A diagonal pass (c) A deep pass and a subsequent back pass

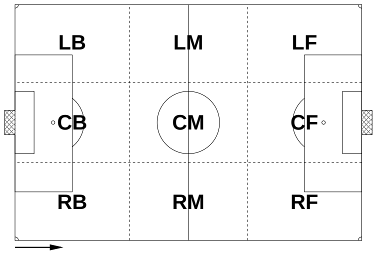
Figure 7.3: Three different ways to build up a play. Dashed lines represent pass-ways and solid lines denote a player's movement. The bold arrow next to the field indicates the direction of play.

i.e. the parameter or goal of a test or operation of a soccer player (agent). A movable *object* in the context of soccer may be a player or the ball. An object is in a current *state*, which includes besides other data the current speed or view direction. Although not explicitly mentioned, a *model of behavior* is assigned to every object, e.g. average or maximum speed or a deceleration rate for the ball as a special case. Additionally, every player needs to hold data about other agents' states. We abstract this by the term *world model*. All this is summarized in the class diagram in Figure 7.4(b).

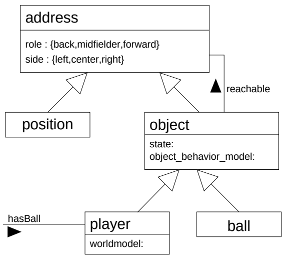
In (Lucchesi 2001, p. ii) only few symbols are introduced that are used throughout the many diagrams in that book: players (in many cases only the team-mates, not the opponents are shown), the ball, passing, movement of the player receiving the ball, and dribbling. Conceptually, all symbols correspond to *actions*, which we abbreviate as $pass$, $goto$, and $dribble$. Since all actions are drawn as arrows starting at some player, naturally two arguments can be assumed: $player$ and $abstract\_position$. $goto(player[LF], region[CF])$ for instance means that the left forward player moves in front of the opposing goal.

Although in most cases this is not explicitly mentioned in (Lucchesi 2001), actions require that certain prerequisites are satisfied, when they are performed. Since our approach aims at a very abstract and universal (league-independent) formalization of soccer, we restrict ourselves to only two tests: possession of ball and reachability. Each of them can be seen as a *predicate* with several arguments: $hasBall$ has the argument $player$ (the ball owner); $reachable$ has two arguments, namely an object and an abstract position.

A pass, for example, presupposes reachability, i.e. it should be guaranteed that the ball reaches the teammate. Note that we already made use of this in Chapter 5.2.2 in the example of playing a double pass. Clearly, the implementation of the reachability test is heavily dependent on the respective soccer league and its (physical) laws. Therefore, at this point, we only give a very general and abstract definition: Object o can reach an address a iff o can move to a and after that



(a) Tactical regions on the field.



(b) Class hierarchy for soccer derived from (Lucchesi 2001).

Figure 7.4: Tactical regions and abstract position hierarchy derived from (Lucchesi 2001). The field is divided into three rows (corresponding to player roles): back (B), midfield (M), and forward (F), and three lanes (sides): left (L), center (C), right (R). An address may be one of the nine regions or player types.

the ball is not in possession of the opposing team. This also covers the case of going to a position where the ball will be intercepted.

7.2.2 Deriving the Specification of Soccer Tactics

The primitive actions we consider here are $goto(player, region)$, $pass(player, region)$, and $dribble(player, region)$. Further we need the action $intercept$ which is a complex action built from the primitive ones. The arguments of the actions are $player$ and $region$ denoting that the particular player should go to, pass, or dribble the ball to the given position. For describing the properties of the world on the soccer field we need the fluents $reachable$ and $hasBall(player)$, among others. The precondition axioms for the actions are:

$$Poss(pass(player, region), s) \equiv hasBall(player)$$
$$Poss(dribble(player, region), s) \equiv hasBall(player)$$
$$Poss(goto(player, region), s) \equiv true$$

For our soccer domain axiomatization, we give successor state axioms for the $ballPos$ function (ball position) and $hasBall$ fluent as examples. We assume that the ball position changes only if we pass the ball to a teammate or dribble with the ball.

$$\begin{aligned} & ballPos(do(a,s)) = b \equiv \\ & \quad \exists player, region. \Big(\big(a = goto(player, region) \land ballPos(s) = b\big) \\ & \quad \lor \big((a = pass(player, region) \lor a = dribble(player, region)) \land b = region\big) \Big) \end{aligned}$$

A player is in ball possession if its position is the same as the ball position. Of course, the player should be located in a certain area around the ball, but for ease of presentation we leave out this detail. If the player passes the ball to another position, the fluent value becomes false.

$$\begin{aligned}&hasBall(player, do(a,s)) \equiv\\&\quad\exists region.\Big(\big(a = goto(player, region) \wedge ballPos(s) = region\big)\\&\quad\vee\big(hasBall(player,s) \wedge \neg\exists region\ a = pass(player, region)\big)\Big)\end{aligned}$$

An Example: Build up Play Revisited

In the following we formalize the examples of building up the play from Figure 7.2, starting with Figure 7.2(a) showing a long pass as first action. There, back player 2 makes a long pass to forward 9, who then passes back to the center midfielder 10, who can make a pass to forward 11, who cuts in deep down-field, as written in (Lucchesi 2001, p. 29). Four teammates are actively involved in this maneuver: back player $p_2 = player[B]$ (whose side need not to be specified), the center midfielder $p_{10} = player[CM]$, and two forwards $p_9 = player[xF]$ and $p_{11} = player[yF]$ on different sides, i.e. $x \neq y$.

Before we are able to formalize the whole maneuver, we have to think about what passing means exactly. A pass from player p to p' requires that p is in ball possession and the ball can be passed to p' beforehand, i.e. the logical conjunction $hasBall(p) \wedge reachable(ball, p')$. Afterwards p' is in ball possession, i.e. $hasBall(p')$. In (Lucchesi 2001, p. 27), three different types of passes are mentioned that can be formalized by additional constraints: (i) long pass with $p.role = B \wedge p'.role = F$, (ii) diagonal pass with $p.side \neq p'.side$, and (iii) deep pass with $p.role < p'.role$ where we assume that the roles (which can also be understood as rows in Figure 7.4(a)) are ordered. With these definitions and constraints for passing, the tactics in Figure 7.2(a) can be described as a sequence of actions in a straightforward manner:

$$\textbf{proc}\ \textit{build-up-play\_4}\ \Big(\big(pass(p_2,p_9); pass(p_9,p_{10})\big)\|goto(p_{11},r)\Big); pass(p_{10},r)\ \textbf{endproc}$$

Recall that subsequent actions (sequences) are marked with semicolon; concurrent, i.e. parallel actions are separated by the symbol $\|$. In addition, $r = region[CF]$ denotes the region in front of the opponent's goal. Since we take the allocentric view from the diagrams in (Lucchesi 2001), we may have parallel actions of different agents (e.g. player 11 running in front of the opposing goal, while player 9 or 10 initiates the pass). Clearly, this has to be turned into an implementation for each agent.

Formalizing the Counterattack

Figure 7.5 depicts a possible move for a counterattack. There, player 8 just captured the ball from the opposing team, dribbles toward the goal, while the forwards (player 9 and player 11) revolve the opponent's defense in order to get a scoring opportunity from both corners of the penalty area while player 10 starts a red herring by running to the center. The white circles represent the opponents. In the original figure (diagram 21 in (Lucchesi 2001), see also Figure 7.2(b)), there are no opposing players as well as no dedicated regions; we inserted them here for illustration purposes.

The specification of the counterattack is given in the procedure $\mathrm{counterattack\_21}$ below.

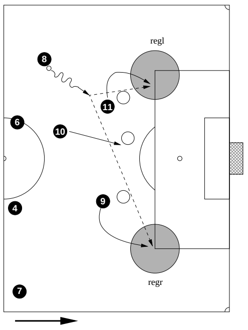


Figure 7.5: Extended diagram 21 from (Lucchesi 2001)

```
**proc** counterattack_21
  $intercept$;
  $startDribble(region[\text{CF}])$;
  **waitFor**($reachable(p_{11}, region[LF]) \vee$
    $reachable(p_9, region[RF]) \vee$
    $\exists x.Opponent(x) \wedge Tackles(x)$);
  $endDribble$;
  **if** $reachable(p_{11}, region[LF])$ **then**
      $pass(region[\text{LF}])$;
  **else if** $reachable(p_9, region[RF])$ **then**
      $pass(region[\text{RF}])$;
  **end**
**endproc**
```

The program is from the view of player 8, that is, all actions and tests are performed by this player. Player 8 gains the ball with an intercept action. He dribbles toward the center (denoted by $region[CF]$)) until either player 11 or player 9 is able to receive the pass or an opponent forces player 8 to do another action (which is not specified in this example). In the specification above, we use the action pair $startDribble$ and $endDribble$ instead of a single $dribble$ action accounting for temporal aspects of that action. Splitting the dribble action into initiation and termination is a form of implicit concurrency, since other actions can be performed while dribbling.

The next step in the presented sequence is a *waitFor* construct. Recall that its meaning is that no further actions are initiated until one of the conditions becomes true, i.e. player 11 or 9 are able to receive a pass in their respective region or an opponent tackles player 8, i.e., an opponent can intercept the ball (go-reachability). Note that during the blocking of the *waitFor* the dribbling of player 8 continues and sensor inputs are processed to update the relation $reachable$ making use of passive sensing as described in Chapter 4.2.2.

Finally, in the conditional we have to test which condition became true to choose the appropriate pass. Note that we do not choose an action in the case of neither player 9 nor player 11 can receive the pass as this would be the matter of another soccer move procedure. The counterattack programs for the other players can be specified similarly.

Reachability

For our formalization of soccer, reachability is central. Reachability strongly depends on the physical abilities of the robot. Besides the physical abilities, the reachability relation has some independent properties. In general, we can distinguish three different reachability relations:

go-reachability: a player p not being in ball possession will reach an address a on the field before any other player: $reachable_{go}(p, a)$ with prerequisite $\neg hasBall(p)$

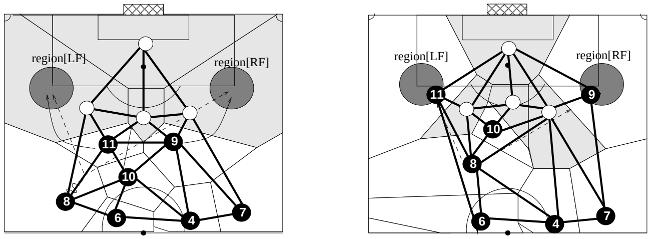
dribble-reachability: a player p being in ball possession is able to dribble towards address a with high probability of still being in ball possession afterwards: $reachable_{dribble}(p, a)$ with prerequisite $hasBall(p)$

pass-reachability: a player p being in ball possession is able to pass the ball b towards address a with high probability of a teammate being in ball possession afterwards: $reachable_{pass}(b, a)$ with prerequisite $hasBall(p)$

To express reachability mathematically, one needs a model which takes the free space between the positions of teammates or opponents into account. One possible model is to use Voronoi diagrams and their dual, the Delaunay triangulation (see e.g. (Aurenhammer and Klein 2000)), as they separate the field into non-intersecting regions and we get a connection graph between the players. Figure 7.6 depicts the Delaunay triangulation and the Voronoi regions for the positions of the players in the counterattack example. The bold lines represent the triangulation, the white and shaded regions correspond to the Voronoi regions of the attacking team and the defending team, respectively. Figure 7.6(b) shows the diagram for the intended situation after player 9 and player 11 have taken their positions in their regions near the goal area. Note that we ignore offside here.

The Voronoi diagram gives us information about which position on the field is closer to which player. In Fig. 7.6(a) we draw the conclusion that the opponent's defense controls the goal area, whereas after the successful counterattack the defense line is penetrated. The triangulation gives information about which player can receive a secure pass. In this particular example there is no connection between player 8 and player 9 and this resembles our intuition that the pass is not secure. The go and dribble reachability is also covered by these diagrams. In Figure 7.6(a) player 9 and player 11, respectively, can test if their target regions are occupied by opponents and they can also test the distance of the defenders to their particular region. With a similar argument the dribble reachability can be expressed. Player 8 can dribble the ball as long as no opponent is in a distance where it can tackle player 8.

With this we can define our $reachable$ relation as a connection between vertices in the Delaunay triangulation. Note that this approach is only one possibility for implementing reachability. The practical experiences made in robotic soccer show that this model is useful as a mathematical



(a) Initial situation. (b) Intended resulting situation.

Figure 7.6: Delaunay triangulation (bold lines) and Voronoi regions (white/gray) for the counter-attack example.

description of reachability not only for pass-reachability, but also for go-reachability and dribble-reachability. In the following section we give examples how free space can be modeled based on Voronoi diagrams.

7.3 A Qualitative World Model for the Robotic Soccer Domain

To implement the soccer moves described in the previous section a qualitative world model is needed. We show one possibility for a qualitatively abstracted world model in this section. First we will introduce several properties of world model data in Section 7.3.1. The representation of the world model depends also on the design paradigm and the goals of the system. Our aim is to formulate soccer strategies and coordinate the behavior of a whole robot team based on a planning approach. Therefore, our world model representation is *explicit*, *global*, *absolute*, and *allocentric*. Our representation is hybrid, i.e. we have both representations, a quantitative and a qualitative one. As the world model predicates computed from sensor readings are quantitative, it does not make sense to ignore them. Thus, the agent or robot is able to make use of both representations.

7.3.1 World Model Categories

The particular choice of what information to include in a robot's world model is not only influenced by the domain, but also by the hardware platform (sensors and actuators), behavior strategy of the team, and the coordination strategy of the team.

With its sensor setting, a robot is limited to perceive only certain aspects of the world. The sensor setup imposes restrictions of the retrievable environment information like the quality of information, how noisy the sensor inputs are, what kind of aspects of the environment are covered by the chosen sensors. Not only the sensory system but also the actuators of the robot restrict the world model. It is not necessary to hold useless information for, say, actions which the robot cannot perform because of its actuators. For example, consider a soccer robot which can only kick the ball in the plane. The world model does not need to keep any information regarding the possibility to perform a loop kick. The sensor and actuator setup therefore restricts the data held

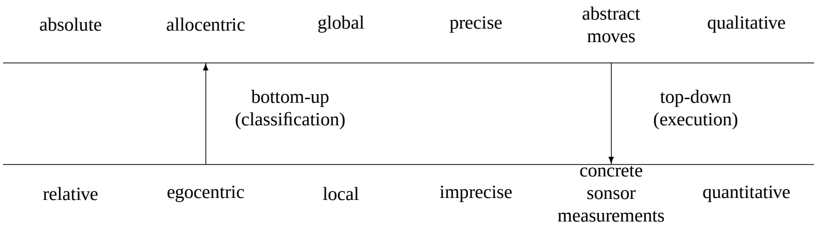


Figure 7.7: Categories of world model representations.

in the world model. This means that for the world model there naturally exists a trade-off between what is desirable, i.e. what the system designer has in mind, and what is possible, i.e. what the chosen hardware and software platform is able to provide. As it is an iterative process to math the both, it is hard to define which kind of information are needed for representing the environment of a soccer robot. Nevertheless, one can define a set of world model predicates which every soccer robots needs. Moreover one can define a set of predicates which are needed to coordinate a team of robots and to come to strategic decisions.

In the following we give an overview about different aspects of world modeling before we derive the qualitative world model for the soccer domain. The different aspects like using a global frame of reference versus using a world model relative to the robot can be transformed to each other. In general, one can distinguish between the categories presented in Figure 7.3.1, which we discuss in the following.

Point of Reference/View. The world model data can have different points of reference. Absolute data have a fixed global point of reference. Usually this is a global coordinate system for all objects or agents. In contrast, the data can be represented with a relative point of reference. For example, another robot or agent can be the point of reference. Then, the data to all the other objects in the world are relative to this particular robot: another robot will collect different data from the same object. A relative representation can always be transformed into an absolute representation by choosing a unique point of reference. Put differently, representations of objects in the world have in general two different perspectives. The one is the global, *allocentric view*. Here, each robot represents the data from a bird's eyes view. In contrast to this view, the *egocentric view* has the robot itself as the center of the world. It means that objects in the world are represented relative to the robot. Consider for example the position of the goals on a soccer field. In an allocentric view the goals have fixed positions, e.g. the center of the goal is located at position $(-5, 0)$. In an egocentric view, the position depends on the position of the robot itself. The position of the goal would be represented as "the goal has a distance of 5 meters ahead and an angle of 0 degree to me". The difference in the representation lies in the coordinate systems of the data. For example, the allocentric view is usually represented with a Cartesian coordinate system while for the egocentric view one would rather choose a polar coordinate system.

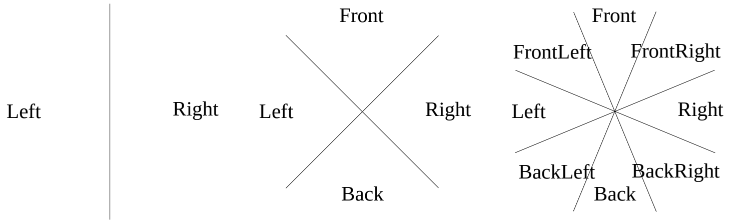
Amount of Information. In this category one can distinguish between *global* information and *local* information. Each robot builds up a local world model which it can acquire through its sensors. If all robots communicate the gathered information about the world and then share their own information with other robots, we speak of a *global* world model. The use of a global world model enhances the quality of the gathered information in general. Unfortunately, it comes at the cost of communication latencies and therefore at the cost of timeliness. The local world model information from all robots must be integrated into a consistent global world model. This costs extra computation times, as inconsistent information due to sensor noise or erroneous calculations in a local world model must be filtered out by appropriate methods. In Chapter 6.5 we compared several known methods for the integration of ball estimates.

Accuracy of Data. The quality of the data of the world model is dependent on the sensor information of the robot. The robot can gather world information only by its sensors which nearly always is noisy. Often, noisy information is captured by attaching a confidence value, which resembles the grade of uncertainty of the information. In some cases, the algorithm used for gathering world model information supplies a mathematical measure about the uncertainty of the respective information, e.g. the position data of the robot which is calculated by a probabilistic algorithm gives a Gaussian distribution of the error (cf. Chapter 6.4), in other cases these measures are empirically found and form a heuristic for the plausibility of the respective world model information. For the decision making of a robot these measures form a trust value about the reliability of the information.

Type of Data. When dealing with sensor systems, nearly all basic data the robot can acquire are quantitative in nature. Consider odometry values gathered by the wheel encoders of the robot. The encoders return a value how much the wheel of the robot has turned in a certain amount of time. Another example is the pose estimation of a robot when a laser range finder is used (cf. Section 6.4). This data is in general quantitative. On the other side of the spectrum are qualitative representations. Qualitative representations are commonly used. For example, for many applications it is sufficient to categorize the speed the robot is driving with in classes like “slow” or “fast”. A complete qualitative world model which we describe in the next section is on the other hand not so common for robotics applications.

Kind of Representation. Each robot acting in any kind of environment needs a world model. In behavioristic approaches like (Brooks 1986) the kind of world model representation is *implicit* on a sub-symbolic level. It means that the system designer does not define predicates or variables to store certain aspects of the world in but derives certain behaviors directly from a sensor input. Nevertheless, this defines a form of world model. In contrast to this, an *explicit* world model can be formulated. Here, each representable aspect of the world gets assigned a predicate which holds the aspect at a certain time of the world. A famous example for this kind of world model representation is the robot system Shakey (Nilson 1984).

Goal of Representation. The decision how the information about the world is represented depends on what the aim of a certain world model information and the general architecture of



(a) level 1 (b) level 2 (c) level 3

Figure 7.8: Different levels of granularity for the orientation from (Hernández et al. 1995)

the system is. In a stimulus-response architecture an implicit, quantitative world model might be sufficient to generate the behaviors of the robot. When one wants to formulate multi-agent strategies making use of a planning system a symbolic, more qualitative representation seems preferable.

7.3.2 Modeling Relative Positional Information

For a qualitative abstraction of positional information we must consider two models: (1) qualitative orientations, and (2) qualitative distances. Combining both yields the qualitative abstraction of the relation of two objects in the plane. Our abstraction of positional information is based on the work of Hernández (1991) Hernández et al. (1995), and Clementini et al. (1997) who present a unified framework for qualitative positional information. The position of a primary object is represented by a pair of distance and orientation relations with respect to a reference object. Both relations depend on a so-called frame of reference which accounts for several factors such as size of objects, different points of view, and so on. Their framework also features basic reasoning capabilities such as the composition of spatial relations as well as switching between different frames of reference. We leave out the mathematical background here concentrating on our adaptations for the soccer domain. For a thorough discussion we refer to the original work Clementini et al. and Clementini et al.

Orientation Relation

The orientation between two objects describes how one object is located relatively to the other. Based on the fundamental observation of how three points in the plane relate to each other, an orientation relation can be defined in terms of three basic concepts: the *primary object*, the *reference object*, and the *frame of reference* which contains the point of view. The point of view and the reference object are connected by a straight line. The view direction is then determined by a vector from the point of view to the reference object. The location of a primary object is expressed with respect to the view direction as one of a set of relations. The number of distinctions made is determined by the level of granularity.

There are different levels of granularity for orientation relations. On the first level the point of view and the reference object are connected by a straight line such that the primary object can be to the left, to the right, or on that line. Thus, the first level partitions the plane into two half-planes. On the second level there would be four partitions, the third level would have eight, and so on. Figure 7.8 depicts the first three levels of granularity for the orientation relation. A well known example of qualitative directions are the points of a compass which correspond to the second level in the simple case: a place can be north, east, south, or west. Sometimes we notice even finer distinctions in terms of intermediate directions such as north-east or south-west. This setting of finer distinctions corresponds to the third level in the above model.

The frame of reference accounts for contextual aspects such as the front side of the reference object. Different frames of reference can be classified into three basic types: they are called (1) intrinsic if the orientation is given by some inherent property of the reference object, (2) extrinsic if a particular orientation is imposed by external factors such as motion, or (3) deictic if the orientation is given by the point of view from which the reference object is seen. Based on the frame of reference there is a 'front' side of the reference object. Independent of the level of granularity there is a uniform circular neighboring structure. In general, at a level of granularity k the set $\{\alpha_0, \alpha_1, \ldots, \alpha_n\}$ denotes the $n+1$ orientation relations where $n = 2^k - 1$. Having a reference object A and a primary object B the orientation of B with respect to A is denoted by $\theta_{AB} = \theta(A, B)$. It can take any of the values mentioned in the set above.

Each orientation has a successor such that $succ(\alpha_0) = \alpha_1$, $succ(\alpha_1) = \alpha_2$, …, $succ(\alpha_n) = \alpha_0$ and a predecessor with $pred(\alpha_0) = \alpha_n$, $pred(\alpha_1) = \alpha_0$, …, $pred(\alpha_n) = \alpha_{n-1}$. In addition each orientation α_i has an opposite orientation which is obtained by applying $(n+1)/2$ times the function $succ$ to α_i. The minimal number of steps needed to get from orientation α_i to α_j along the circular neighboring structure is called range. The range between opposite orientations is $(n+1)/2$. Two orientations are called orthogonal if the range between them is $(n+1)/4$ (except for the basic level 1). Each orientation has two orthogonal orientations to it.

Distance Relation

Similarly, the distance relation requires three elements as well: the primary object, the reference object, and the frame of reference.

In a 2D metric space, the distance between points is defined by the following three axioms

1. $dist(P_1, P_1) = 0$ (Reflexivity),
2. $dist(P_1, P_2) = dist(P_2, P_1)$ (Symmetry), and
3. $dist(P_1, P_2) + dist(P_2, P_3) \leq dist(P_1, P_3)$ (Triangle Inequality).

Conventional concepts of distance normally rely on coordinates. The distance between points $P_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ in a n-dimensional vector space can be expressed in terms of the Minkowsky L_p-metric (Preparata and Shamos 1985):

$$d_p(P_1, P_2) = \Big(\sum_{j=1}^{n} |x_{1j} - x_{2j}|^p\Big)^{1/p}$$

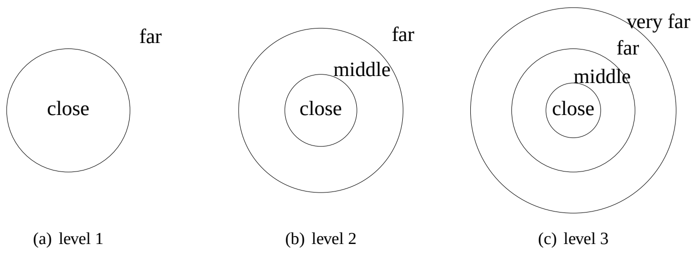


Figure 7.9: Different levels of granularity for the distance relation

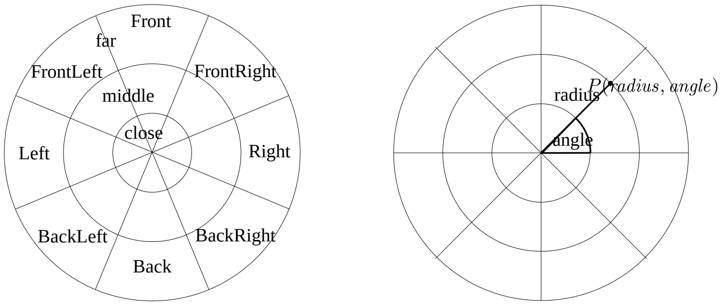
Well-known examples are for instance, the Manhattan distance which is defined by the L_1-metric or the Euclidean distance which is defined by the L_2-metric.

Similar to the orientation relation we can distinguish distances at various levels of granularity. An arbitrary level n of granularity with $n+1$ distinctions yields the set $Q = \{q_0, q_1, \ldots, q_n\}$ of qualitative distances. Given a reference object RO these distances partition the space around RO such that q_0 is the distance closest to RO and q_n the one farthest away. The qualitative distance between the reference object and a primary object, both belonging to a set O of objects, is a function $d : O \times O \to Q$. Thus, the distance between the primary object A and a reference object B then is $d_{AB} = dist(A, B)$, where $dist$ denotes the distance from A to B.

Next, we have to choose an appropriate distance system for our domain. We choose a *homogeneous* partitioning, i.e. our distances measures follow a recurrent pattern like $q_1 = 2 \cdot q_0, \ldots$, instead of using equally spaced distance ranges. That is not only because the perception of objects and situations farther away is less accurate but also because of a temporal aspect. The robot needs more time to get to positions farer away, and objects there are more likely to change their positions before it gets there. Each distance interval grows by factor s w.r.t. its direct predecessor. As we already stated there is a maximal distance d_{max} due to the fixed size of the pitch. We compute its value from the diagonal of the rectangle formed by the playing field and its additional boundary border. $d_{max} = \sqrt{(w_{field} + 2 \cdot w_{border})^2 + (l_{field} + 2 \cdot l_{border})^2}$. The distance system we use is parametrized with the level of granularity n chosen. At level n there are $n+1$ distance relations. We introduce another distance relation called out_of_field which subsumes all distances that are greater than d_{max}. The maximal distance can then be represented by $d_{max} = \|\delta_0\| + \|\delta_1\| + \ldots + \|\delta_{n-1}\|$. It has to be subdivided into n intervals with growing size. Each distance relation is s times larger than its predecessor. Thus, $\|\delta_i\| = s \cdot \|\delta_{i-1}\|$ or $\|\delta_i\| = \|\delta_0\| \cdot s^i$. d_{max} can then be expressed as

$$d_{max} = (n-1)\|\delta_0\| \cdot \sum_{i=0}^{n-1} s^i.$$

with $\|\delta_0\| = d_{max} / \sum_{i=0}^{n-1} s^i$. For simplicity we have chosen a strict interpretation of our distance relations. This leads to sharp boundaries for our distance intervals. Thus, we need exactly n values to represent $number\ of\ distances$ many intervals δ_i. These intervals are then constituted



(a) The combination of the distance and the orientation relation

(b) A point p defined in polar coordinates

Figure 7.10: The combination of distance and orientation relation compared to the polar coordinate system.

as follows:

$$[0, a_1[, [a_1, a_2[, \ldots, [a_{n-1}, a_n[, [a_n, +\infty].$$

To obtain the qualitative distance for a quantitative distance value we simply determine to which of the intervals it belongs.

Combining Orientation and Distance

In order to describe (and reason about) positional information we now investigate the combination of the distance and the orientation relation. Putting together the two relations presented above, allows us to represent the location of a primary object B relatively to a reference object A. From a quantitative point of view, the combined description of a position can be seen as the representation of a point in polar coordinates. The two-dimensional polar coordinate system involves the distance from the origin and an angle. A point p in polar coordinates is defined by the distance r from the origin to the point and the angle φ measured from the horizontal x-axis to the line from the origin to p in the counterclockwise direction. Thus, the position of a point p is described as (r, φ). This description corresponds to a combination of the distance relation and the orientation relation which we presented in the previous sections. We depict an illustration in Figure 7.10.

Most of the time, we use the Cartesian coordinate system to represent a position. As we have just seen, the combination of our qualitative distance and orientation relation corresponds to the definition of a point in polar coordinates. Fortunately, there is an easy way to switch between the two systems. If we choose a Cartesian coordinate system with the same origin as with the polar coordinates and if we have the x-axis in direction of the polar coordinates, we have the following formulas for the transformation between the two systems: $x = r \cdot \cos(\varphi)$, $y = r \cdot \sin(\varphi)$, and $r = \sqrt{x^2 + y^2}$, $\varphi = \arctan \frac{y}{x} + \pi \cdot u_0(-x) \cdot \mathrm{sgn}(y)$ where u_0 is the Heaviside function[2] with

[2]The Heaviside function, sometimes called the unit step function, is a discontinuous function whose value is zero

$u_0(0) = 0$ and sgn is the signum function. Here, we use the functions u_0 and sgn as logical switches instead of a distinction of different cases.

By this we can transform the distance and orientation relations to Cartesian coordinates which we use in our system. Beside the relative positional information described so far positions are also used globally. We are going to investigate a possible representation approach for this in the next section.

7.3.3 Modeling Semantic Regions

As we already observed in our analysis of soccer theory one of the qualitative concepts applied frequently is that of semantic regions on the playing field. These regions are often used as tactical positions corresponding to player roles. There are three zones: back, midfield, and forward. Furthermore, there are three sides: left, center, and right. Combining the two partitions above results in a subdivision of the field into nine regions. This seems to be enough to cover the role description task (cf. Section 7.2.1). Although, when specifying soccer moves for a team of autonomous soccer agents, it may be necessary to have a finer distinction. Thus, to retain flexibility we aim at building a representation framework that is adjustable to different requirements.

A well-known approach to qualitative representation of positional information was proposed by Freksa and Zimmermann (1992). It bases on a directional orientation information. The approach is motivated by considerations on how spatial information is available to humans and to animals: directly through their perception. Thus, cognitive considerations about the knowledge acquisition process build the basis here.

Qualitative orientation information in two-dimensional space is given by the relation between a vector and a point. The vector consists of a start point A and an end point B. It represents the orientation of a possible movement. Suppose a line through A and B, and consider two lines each going orthogonally through A and B. These three lines form an orientation grid which has the form of a double-cross. Different positions of an additional third point C can be described as follows. In relation to the line through A and B a point C can either be to the left-hand side of this line, straight on this line, or to the right-hand side. The lines through A and B allow for further distinctions of the position of point C. It can either be in front of the line through B, just on the orthogonal line through B, neutral in between the two orthogonal lines, on the line through A, or back, that is, behind the line through A. Altogether this leads to 15 different orientation relations. Figure 7.11 depicts the grid presented above and the iconic representations of the different possible positions of a point C in this grid.

Reasoning with these relations is possible through four operations: *inversion*, *homing*, *shortcut*, and *composition*. Freksa (Freksa 1992) uses two different types of composition, one of them named coarse composition and the other named fine composition. While the coarse version produces very imprecise results the fine version is enhanced with rudimentary distance information. These supplementary distance information allow for a more exact composition.

The approach presented in (Freksa 1992) and (Freksa and Zimmermann 1992) is quite intuitive, because it is based on human cognition. However, any relation is based on a vector between

for negative arguments and one for positive arguments. The value of $u(0)$ can be defined freely; it is often indicated as an index to u, that is u_0 in our case.

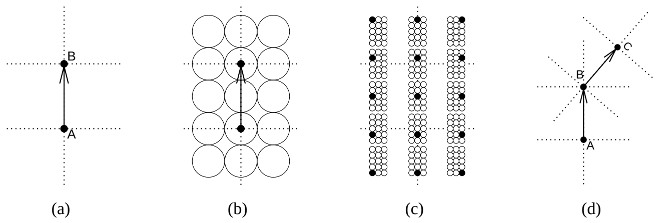


Figure 7.11: The double-cross calculus by (Freksa and Zimmermann). (a) shows the orientation vector from A to B. (b) shows the possible position of an additional point C. (c) shows the iconic notation for each of the possible positions of C. (d) A possible composition of two orientation relations.

two points, which cannot be taken for granted in the context of our work. Spatial settings in soccer not always involve a movement but they also describe static situations. Furthermore, we need to represent inherent spatial characteristics of a player in terms of qualitative predicates like the view direction. These inherent traits also do not provide a motion vector to which we could apply Freksa's model. Thus, it is not always applicable for us. Nonetheless, we can use the orientation grid to abstract from the quantitative values.

Inspired by the work of Freksa (1992) and Freksa and Zimmermann (1992), we develop our model of semantic regions for soccer play. The orientation grid used for the representation of orientation alignment bases on a (movement) vector going from a point A to a point B. Although we do not have an explicit movement in the context of global positioning on the playing field we can regard the direction of play as a vector. From a tactical point of view the focus of a game is to advance from a defensive situation in a team's own half to an offensive one in the opponent's half. Thus, we can take the center of each team's half as the start and end point of our imaginary vector. If we place this vector onto the playing field Freksa's orientation grid yields 15 regions. The regions and their derivation are depicted in Figure 7.12.

The regions originating from embedding Freksa's orientation grid onto the playing field are separable by the field's two dimensions. We can carry out a subdivision along the field length and width. This procedure results in *zones* for the length of the field and in *sides* for the width. To preserve the importance of the center of the field w.r.t. its tactical meaning, we take it as the center for both, the zones and the sides of the field. In order to have a central position on the field we chose both the number of zones and the number of sides to be odd.

There is no obvious reason not to choose equidistant intervals for our representation of regions. The reason is that the regions of the playing field do not differ in terms of size or shape seen from a global point of view. Basically they form an abstraction grid, which consists of equally formed sectors subsuming the set of positions of a certain area. Since we want our regions to be as flexible as possible we parametrize the model by the desired number of distinctions for each dimension of the pitch. Hence, we can determine the length and the width of each region by $\|region\|_x = w_{\text{field}}/number\ of\ sides$ and $\|region\|_y = l_{\text{field}}/number\ of\ zones$.

The system sketched above consisting of zones and sides having its center in the middle of

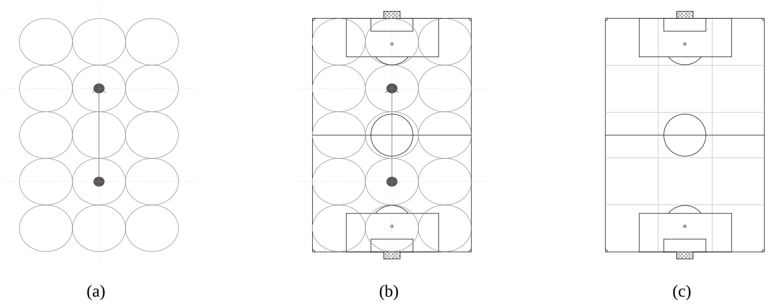


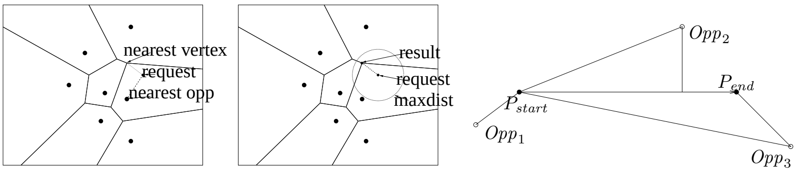
Figure 7.12: Semantic regions on the playing field. Figure (a) shows the orientation grid taken from (Freksa 1992). Figure (b) shows the grid embedded into a soccer field. The resulting semantic regions on the playing field are shown in Figure (c).

the playing field roughly corresponds to a coordinate system. Zones and sides form perpendicular axes, where the zones correspond to the x-axis and the sides correspond to the y-axis of a Cartesian coordinate system. We still can specify an object's position in a coordinate-system-like manner, but by using zones and sides we achieve a qualitative description. In the following section we are going to elaborate on possible applications of the qualitative approach to regions we just presented.

7.3.4 Reachability, Free Space, and More

In Section 7.2.2 we already discussed reachability as a central relation for soccer. As a possible mathematical model for reachability we have chosen and are making use of Voronoi diagrams.

The notion of free space is another important aspect, which can frequently be found in the description of spatial settings and tactical patterns in soccer. The term free space denotes an area which is not occupied by any of the players of the opposing team. Consider a Voronoi diagram being constructed from all opponent's positions. A Voronoi vertex (a crossing of Voronoi edges) in the diagram constructed from the opponents' positions determines a point with maximal distance to all surrounding opponents. This means that this point is the most free point in that particular region. To acquire these free regions we provide two different methods. First, we consider a classification request. Given a point on the playing field, we can ask how 'free' this point is. To answer such a request we compute the point's distance to the nearest point site (the nearest opponent) in the opponent's Voronoi diagram as well as its distance to the nearest Voronoi vertex. The ratio between these two values is a good criterion on how 'free' the given point is. Of course, the ratio is not always a sufficient indicator since it does not reflect the absolute distance to an opponent. Therefore, we additionally require a minimal distance which has to be exceeded for a position to be classified as being free. Secondly, we can answer inquiries for free point positions. Most of the times it is reasonable to specify a region of interest, in which to search for a free position. For simplicity we assume that this region of interest is specified by a position in the coordinate system of the pitch (along with a maximal distance limiting the search). Given this



(a) Classification request (b) Point request (c) Pass-way vacancy

Figure 7.13: Free space and pass-way vacancy.

query, we determine the nearest Voronoi vertex to the position. If the distance between the query point and the vertex is less than the maximal distance we return the vertex' position. Otherwise, we return the position of a point lying on a line starting at the query point going into the direction of the nearest Voronoi vertex. We depict an illustration for both queries in Figure 7.13.

Within the course of a soccer game it is a vital information whether or not ones team is in possession of the ball. We can provide this information by exploiting the structure of Voronoi diagrams. The simplest way to answer the question of ball possession is to check if the ball is located in the Voronoi region of a player who belongs to ins's own team. This is not always correct. For example, if the player whose Voronoi cell the ball belongs to is not facing the ball, it might be the case that another player who has a greater distance to the ball but who is directly facing it can reach it more quickly. It is, however, possible to take this additional information into account and to refine the predicate accordingly.

As an additional qualitative predicate of particular interest in the soccer context we further consider something we call pass-way vacancy. We denote a qualitatively abstracted classification of the amount of space available along a potential pass-way by this predicate. That is to say, we classify the degree of exposure of a line segment going from point P_{start} to point P_{end} by examining possible sticking points or points of interception. We derive our classification by considering a ratio on how likely an interception is. Consider a straight line from P_{start} to P_{end}. We compute the minimal distance of each opposing player to this line, which is either the length of a line perpendicular to the pass-way or the distance to the pass-way's nearest end point. Further, we compute the distance from each opponent to the starting point of the passageway. Then, we calculate the ratio between these two values. That is to say, we determine if the opponent is so close to the pass-way that it can intercept a ball passed along the pass-way. Figure 7.13(c) shows an illustration of the calculations performed when determining a PasswayVacancy.

As we already pointed out in Chapter 7.2 another information that is beneficial for the specification of tactical patterns can be provided by the *unmarked* predicate. It is used to state whether a player is free or covered by an opponent. This information is essential, for instance, to decide if a pass to this player makes sense or not. A simple way to answer this question is to calculate if there are any opponents within a certain distance to the player desired.

7.4 Using State Space Abstractions for Soccer: Generating a DT Plan Library

Having constructed a world model consisting of qualitative predicates as presented in the previous section, the question is whether we gain expressiveness for formulating high-level tasks of the robot or agent, or if we can solve problems which we could not be solved before. In Chapter 4 we introduced an approach to use macro actions in a decision-theoretic context. Recall that our proposed method was to use ordinary value iteration until the values for all states of the macro action converged. We were therefore restricted to problems where we could determine the state space for the macro action. This precluded the use of decision-theoretic macro action in more complex domains like robotic soccer.

In this section we present the idea of using macro actions taken a step further. The idea is rather simple: we first calculate an abstract policy without filling in the decisions the forward search algorithm takes. With this, we calculate an abstract policy consisting of all possibilities the input program leaves open. This is done off-line. Each of these so-calculated macros are instantiated at run-time in the particular world situation the agent is facing. Finally, we store the outcomes of our macro action in a plan library so that the agent, if it is facing the same world situation again, exactly knows which of the possible macro actions might be the best. To make this idea work for robotic soccer we need to abstract the soccer state space with the just described qualitative world model. In the following we show our approach to build up an abstract plan library and give two examples from the Simulation League revealing that the approach together with the qualitative world model allows it to define macro actions, and that these macros turn out to be beneficial w.r.t. to computation times.

7.4.1 Solving Decision-theoretic Plans in an Abstract Way

Recall that READYLOG's forward-search algorithm calculates a policy from an input program. By taking into account all possible outcomes of a stochastic action, nondeterministic choices in the input program are optimized away. For each agent's choice point the forward-search algorithm selects the best alternative, for each nature's choice point given by stochastic actions a conditional branching over the possible outcomes is introduced. The calculations of values and probabilities rely on the current situation term, i.e. the reward is given w.r.t. a particular situation. Therefore, the algorithm can decide optimal choices at choice points.

The idea for generating a plan library is now the following. In a run of the forward-search we do not calculate explicit numeric values for the reward function but keep it as terms. Basically, we store the whole computation tree for a respective input program. Later, when instantiating a plan from the plan library, we can establish the optimal policy, the values and probabilities of all outcomes of the policy. By this, we make use of the trade-off between space and time. It turns out that with re-instantiating the abstract terms and re-evaluating the optimal choices one can save a significant amount of computation time compared to the on-line interpretation of a decision-theoretic program. For the implementation of the calculation of policies in an abstract way, we have to modify several $BestDo$ predicates in such a way that choices are not taken, but all possible continuation policies are calculated. When calculating the possible continuation policies for a conditional **if** φ **then** a_1 **else** a_2 **endif**, for instance, we have to calculate both branches, where φ and $\neg\varphi$ holds. To do so, we introduce a new predicate $BestDoM$. $BestDoM$ is basically the same as $BestDo$, despite that we replace the calculation of the value by an abstract value term, and that we calculate all possible continuation policies. To give an example, we show the macro for a stochastic action.

$$
\begin{aligned}
&BestDo(A;p,s,h,\pi,v,pr) \overset{def}{=}\\
&\quad \exists P.procmodel(A,P) \wedge\\
&\qquad \exists \pi',v',pr'.BestDoAux(P,p,s,h-1,\pi',v',pr')\\
&\qquad \pi = a;\pi' \wedge v = +(-(reward(s),cost(a,s)),v') \wedge pr = pr'
\end{aligned}
$$

Basically, the difference to the predicates introduced in Chapter 4 (Eq. 4.1 on page 85) is that instead of calculating the value with the formula $v = reward(s) + v' \wedge pr = pr'$ as is done in the original $BestDo$ predicate, we keep the term $v = +(-(reward(s),\ cost(a,s)), v')$. Further, we do not check whether the stochastic procedure is possible or not. The reason for this change is that we cannot decide whether or not the respective outcome action is possible, as we are not given a concrete situation where we could evaluate the predicate. This must be checked when executing the so-calculated policy. Again, the value as well as the probability of success is calculated as a term depending on situation s, not as a concrete value. The respective auxiliary predicates for the stochastic action are:

$$
\begin{aligned}
&BestDoMAux(sprob(\{(n_1,p_1,\varphi_1),\ldots,(n_k,p_k,\varphi_k)\},p,s,h,\pi,v,pr) \overset{def}{=}\\
&\quad \exists \delta,s'.trans^*(n_1,s,\delta,s') > 0 \wedge\\
&\qquad \exists \pi',v',pr'.BestDo(p,s',h,\pi',v',pr') \wedge\\
&\qquad \exists \pi'',v'',pr''.BestDoAux(\{(n_2,p_2,\varphi_2),\ldots,(n_k,p_k,\varphi_k)\},p,s,h,\pi'',v'',pr'') \wedge\\
&\qquad \pi = \textbf{if } \varphi_1 \textbf{ then } \pi' \textbf{ else } \pi'' \textbf{ endif} \wedge\\
&\qquad v = +(v,\cdot(v_1,prob(n_1,a,s))) \wedge pr = +(pr',\cdot(p_1,prob(n_1,a,s)))
\end{aligned}
$$

$$
\begin{aligned}
&BestDoMAux(sprob(\{(n_1,p_1,\varphi_1)\}),p,s,h,\pi,v,pr) \overset{def}{=}\\
&\quad \exists \delta,s'.trans^*(n_1,s,\delta,s') > 0 \wedge Final(\delta,s') \wedge\\
&\qquad \exists \pi',v',pr'.BestDo(p,s',h,\pi,v,pr) \wedge\\
&\qquad v = \cdot(v',prob(n_k,a,s)) \wedge pr = \cdot(pr',prob(n_k,a,s))
\end{aligned}
$$

As we have stressed before, the difference of our $BestDoM$ definitions compared to the original $BestDo$ is that (1) all possible continuation policies have to be calculated, and (2) the value as well as the probability of success are handed over as terms. Later, when the policy is instantiated in a concrete situation, we could evaluate the value and probability term to get the real numbers. To illustrate this again, we come back to our well-known maze world example from Chapter 4. We give the abstract value for the agent calculating one step of the (simplified) policy to leave room 1 from the start position "S" through the northern door:

$$\begin{aligned} v \quad = \quad & +(-(reward(do(go\_up, s)), cost(do(go\_up, s))), \cdot(prob(go\_up, det\_up, s), \\ & \quad reward(do(det\_up, s)), \cdot(prob(go\_up, noop, s), reward(do(noop, s)))), \\ & +(-(reward(do(go\_right, s)), cost(do(go\_right, s))), \cdot(prob(go\_right, det\_right, s), \\ & \quad reward(det\_right, s)), \cdot(prob(go\_right, noop, s), reward(do(noop, s)))), \ldots \end{aligned}$$

Similarly, the first step of the abstract policy to leave room 1 looks like

$$\begin{aligned} & (\mathtt{poss}(\mathtt{go\_up}, \mathtt{S}) \rightarrow ((\mathtt{has\_val}(\mathtt{pos}, \mathtt{V}_6, \mathtt{S}), \mathtt{V}_6 = [\mathtt{V}_{11}, \mathtt{V}_{12}]), \mathtt{V}_{14}\ \mathtt{is}\ \mathtt{V}_{12} + (1), \mathtt{V}_{13}\ \mathtt{is}\ \mathtt{V}_{14}), \\ & \quad \mathtt{V}_{111} = [\mathtt{go_up}, \mathtt{if}(\mathtt{pos} = [\mathtt{V}_{11}, \mathtt{V}_{13}], [], [\mathtt{if}(\mathtt{pos} = [\mathtt{V11}, \mathtt{V12}], [], [])])]; \mathtt{V111} = [], !), \\ & (\mathtt{poss}(\mathtt{go\_right}, \mathtt{S}) \rightarrow ((\mathtt{has\_val}(\mathtt{pos}, \mathtt{V}_{19}, \mathtt{S}), \mathtt{V}_{19} = [\mathtt{V}_{24}, \mathtt{V}_{25}]), \mathtt{V}_{27}\ \mathtt{is}\ \mathtt{V}_{24} + (1), \mathtt{V}_{26}\ \mathtt{is}\ \mathtt{V}_{27}), \\ & \quad \mathtt{V}_{110} = [\mathtt{go\_right}, \mathtt{if}(\mathtt{pos} = [\mathtt{V}_{26}, \mathtt{V}_{25}], [], [\mathtt{if}(\mathtt{pos} = [\mathtt{V}_{24}, \mathtt{V}_{25}], [], [])])]; \mathtt{V}_{110} = [], !), \\ & (\mathtt{poss}(\mathtt{go\_down}, \mathtt{S})\mathtt{to}((\mathtt{has\_val}(\mathtt{pos}, \mathtt{V}_{32}, \mathtt{S}), \mathtt{V}_{32} = [\mathtt{V}_{37}, \mathtt{V}_{38}]), \mathtt{V}_{40}\ \mathtt{is}\ \mathtt{V}_{38} - (1), \mathtt{V}_{39}\ \mathtt{is}\ \mathtt{V}_{40}), \ldots \end{aligned}$$

Remember that `has_val` is a predicate to evaluate a fluent, and `poss` checks whether the precondition of an action holds. For ease of presentation, the example is simplified as the basic actions only have one failure case, namely a $noop$ action where the agent will stay on the same position. As one can see from the small examples above the policies grow large even for small toy problems. Even if the optimal solution to leave room 1 is encoded as an option, i.e. as a macro action, the option cannot be represented in a compact way as each possible outcome for each action has to be provided when calculating abstract policies and values. Though, as we will show in Section 7.4.3 one can buy the larger space consumption by less computation time.

7.4.2 Generating a DT Plan Library

In the previous section we presented our idea of solving decision-theoretic problems in an abstract way implemented by a new predicate $BestDoM$. Independent from a particular situation, this predicate generates, from a given program, a list of abstract policies with a corresponding abstract representation of the value function. In a given situation, this abstract representation can be easily re-instantiated with numerical values, which makes it possible to choose and execute the highest valued abstract policy. Our idea is now to use this abstract plan instead of on-the-fly decision-theoretic planning and to build a DT PLAN LIBRARY, a library that contains policies that already were calculated and used before. These stored policies are then provided for re-use if the agent comes to similar states again.

This is the basic idea of our new options approach: an option is essentially a READYLOG program with a solve statement which encodes the contents of the option (similar to the $navigate$ program in Section 4.2 on page 73). This means that with this program the partition of the state space of the option is induced. The states reachable with the program form the state space of the option. Note that we use the term state here and use it similar as is done in related action formalisms like FLUX (Thielscher 2005) (cf. also Chapter 2.1.5, page 18). A state refers to a finite (sub-)set of all fluents which are needed to solve the option. A finite subset of instantiated fluents form the state representation for the option and must be given by the user. This state description must contain enough information to evaluate tests and action preconditions mentioned in the input program from which the option is calculated. For the maze domain our state representation comprises only the location fluent. It is sufficient for navigating between the rooms in the maze. In the soccer domain, on the other hand, we need to find the fluents which are important for the option. Obviously, is too much to take the positions of all 22 players on the field into account for a free kick option. Only the players near the ball are of importance. In the MDP theory using logical literals to describe the state of an MDP is also referred to as a factored representation of the state space.

To calculate a policy for an option the following steps have to be conducted:

1. *Off-line pre-processing*
   (a) Calculate an abstract policy for each solve statement occurring in the behavior specification.
   (b) Replace each solve statement with its abstract policy in the specification.
2. *On-line execution*
   (a) Look up the policy, value, and probability of success for the option in the DT PLAN LIBRARY.
   (b) If the option is not contained in the library, instantiate the option in the particular situation and store the value and the probability of success together with the current world state in the library.

In the off-line part we pre-process each occurrence of a solve statement in our agent high-level specification with the $BestDoM$ predicates shown in the previous section. As the result we obtain for each solve statement an abstract policy as presented before.

When executing an option in a particular situation we first query our DT PLAN LIBRARY if for the current world situation an instantiated policy for the option currently to be executed exists. If so, we simply take this policy from the library and execute it. If there does not exist a policy for the option in the current world situation, we have to generate it. We take the situation independent abstract policy for the option and substitute the situation terms with the actual situation. Similarly, we evaluate the value and success probability of the option given the current world situation. With a particular situation we can re-evaluate the precondition axioms of actions, if-conditions, and nondeterministic choices of the abstract policy and obtain one fully instantiated policy which is the same as if we would have calculated it on the fly. To gain computation speed for the next time when the agent wants to execute the option in this particular situation, we store the fully instantiated policy, the value, and the success probability together with the world state. Thus, the next time the option is to be executed in the very same situation, we simply look up the policy without the need to calculate anything at all.

The on-line execution is illustrated in Algorithm 7.14 on page 209. The predicate `getState` calculates the current world state based on fluent values as described above. The predicate `get_bestPolicy` performs the look-up operation, the predicate `evaluate` assesses the abstract plan tree returning a fully instantiated policy π_s, which is then executed with `execute`(π_s). The

```
getState;
while $\varphi_m$ do
    if DT PLAN LIBRARY has entry for current state $s$ then
        get_bestPolicy(s,DT PLAN LIBRARY,$\pi$);
        execute($\pi_s$);
    else
        evaluate($s$,AbstractValues,$\pi_s$);
        execute($\pi_s$);
        store($(s, \pi_s, v, pr)$, DT PLAN LIBRARY);
    end
    execute($a_{sense}$);
end
```

Figure 7.14: The algorithm executed by a macro-action.

`store` predicate saves the instantiated policy, the value, and the success probability together with the current world situation in the DT PLAN LIBRARY for the next time it is needed. The action a_{sense} is a sensing action which is executed to sense the actual state the agent is in, when trying to execute the option. The logical formula φ_m is a condition which checks if the option is executable. This condition can be viewed as a precondition for the option. This precondition is part of the specification of the option and must be provided by the user.

The main difference to the original options approach as presented in Chapter 4 is that we do not use standard value iteration to calculate the behavior policy, but make use of the forward-search algorithm which is also used for decision-theoretic planning in the framework. The advantage is that with our new approach we gain flexibility in the sense that the input program from which the option is calculated determines the state space of the option. Determining the sub-tasks in toy domains like the maze domain is rather easy (leaving rooms through doors). But in realistic domains like robotic soccer this tasks is not that easy and therefore it helps to define the option by the input program.

7.4.3 Experimental Results from the Simulation League

As an extension to options in READYLOG we propose macro actions based on an abstract set of precomputed policies and a DT PLAN LIBRARY, with the aim to speed up the agent's behavior in contrast to on-the-fly decision-theoretic planning. In this section we present the results we obtained by testing our approach in a continuous real-time domain. The presented results are from the 2D Soccer Simulation league. The results shown here are rather proof-of-concept results. Further investigations have to be taken especially w.r.t. scalability aspects of our proposal.

We define two macro actions to test the applicability of our approach in an environment such complex as robotic soccer. A building block of being able to apply our new macro action to soccer is the use of the qualitative world model. The agent's position, for example, can be the coordinate $(12.0226375902, 5.847583745)$. The probability, that the agent comes to exactly this position twice is nearly zero. Building equivalence classes over ranges of positions, for instance, ensure that the agent meets a position on the field more than once. Otherwise, it would preclude a re-

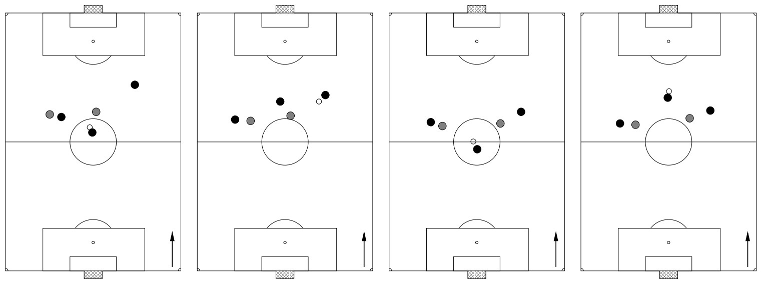

(a) First setting for out-playing opponents. (b) The macro-action chooses to pass and go. (c) Second setting for out-playing opponents. (d) The macro-action chooses to dribble.

Figure 7.15: Outplay Opponent; (a)-(b): “pass and go”.(c)-(d): “dribble”.

use of the policies stored in the DT PLAN LIBRARY. Nevertheless note that even with using the qualitative world model together with the options approach as presented in Chapter 4, we could not formulate the behaviors of the soccer agent as a macro action. That is because in the original approach we have to enumerate the whole state space which is fairly possible for a soccer action.

The first macro action is designed to *outplay opponents* as shown in Fig. 7.15(a) and 7.15(b). Facing attacking opponents, the ball leading agent either dribbles or passes the ball to a teammate. If the macro action chooses the pass, the agent afterwards moves to a free position to be a pass receiver again. The second action aims to *create a good scoring opportunity* to shoot a goal. The agent in ball possession can dribble with the ball if the distance to the opponent’s goal is too far. Near the goal the agent can shoot directly to the goal or pass to a teammate that is in a better scoring position. We compared macro actions with DT planning for the game settings we created as test scenarios. Besides the computational behavior, we also investigated the action’s choices in specific states. We want to remark that by using the proposed state abstraction we commit to some sort of bias in the representation of the environment that is reflected in the state variables. In fact, these variables have to ensure that the macro action correctly chooses a policy fitting the game setting. Situations which require different policies have to be represented by different states. This can be handled by the granularity of the qualitative world model. The provided grid is adjustable by hand and can be created fine-grained enough to distinguish significant changes in the game setting. In the first setting depicted in Fig. 7.15(a) the ball leading agent directly faces an opponent. In this state the macro action evaluates a pass to the uncovered teammate and a subsequent move action as best policy (Fig. 7.15(b)). In the second setting given in Fig. 7.15(c), both opponents cut the possible pass-ways for the ball leading agent to its teammates. The state representation ’detects’ this difference and evaluates a new policy. The agent dribbles towards the opponent’s goal (Fig. 7.15(d)). Qualitatively there is no difference in the behavior between the on-the-fly DT planning and the macro approach, as both rely on the same READYLOG input programs. What can be observed, though, is that the macro action approach needs less time to come to a decision.

We considered three strategies: (a) using DT planning to cope with the task, (b) using the macro action, but only by evaluating a policy in each step[3], and (c) using the macro action with the DT PLAN LIBRARY that was generated in the previous step. We conducted 20 iterations per setting. Using the planning approach the agent needed 0.1 seconds on average to calculate a policy. With the evaluation strategy (b) only 0.08 seconds are needed. This is a speed-up compared to planning of about 20 %. The time for off-line computations in this example was about 0.02 seconds for each macro. Even taking this pre-processing time into account our macro approach yields reasonable speed-ups. The pre-processing time naturally increases with the complexity of the macro action model. But as this time does not need to be spent on-line this off-line computation time can be justified. The macro action based on the DT PLAN LIBRARY clearly outperforms DT planning. In each test-run, for both macro actions, the executing system constantly returns the minimum of measurable time of 0.01 seconds for searching the best plan in the DT PLAN LIBRARY. In fact, this is a mean time saving of over 90 %. In tests in the ROBOCUP 3D simulations which we conducted in another test scenario these savings showed an impact on the quality of the agent's behavior in real game situations. Caused by a reduced reaction time the agent showed fewer losses of the ball during the game. Moreover the team created more opportunities to score than in test runs without using the macro actions. This goes along with a space consumption of about 10 kB for each defined macro action. Our examples reflect the task to find a suitable policy in a split second for the ROBOCUP domain. In practice, our policies are quite short, since computing larger policies is not (yet) reasonable as the world changes unpredictably for larger horizons. In our application examples the larger space consumption does not play a major role. For more complex macros the space consumption of the exponentially growing computation tree has to be further investigated. It is due to future work to examine how well our method scales.

In this section we showed how a significant speed-up can be gained for calculating policies with an alternative macro-action approach. The basic idea is to calculate and store all possible calculation branches of the used forward-search algorithm together with their values and probabilities in an abstract fashion. When the agent faces a world situation which it already encountered before, it can simply draw on a previously calculated policy, re-instantiate it with the current situation it is in and gains an optimal policy instantaneously. While previous proposed approaches like those presented in Chapter 4 also resulted in an exponential speed-up for toy domains, they fail for continuous domains like robotic soccer. The reason is that the method presented before relies on an explicit state enumeration for calculating the result of an option. Even when soccer state space abstractions (the qualitative world model) are used the problem of deciding the states of an option remains. With our second approach of a plan library on the other hand, this problem is no longer apparent. Which states belong to an option is implicitly given by the input READYLOG program. Nevertheless, the original options approach in READYLOG can be easily modeled equivalently with our plan library. The appeal of the plan library lies, for one, in the speed-up of the agent's decision making, for another in it simplicity. By simply exploiting the well-known trade-off between space and time we can speed up decision-making in READYLOG significantly.

[3]Each policy evaluated in this step is stored in the DT PLAN LIBRARY, so we can use this stored knowledge in the next step (c).

7.5 Summary and Related Work

In this chapter we showed that READYLOG cannot only be used as an agent programming language, but is with its expressiveness well-suited for formalizing agent behaviors in general. From human soccer theory literature we derived basic primitives needed to encode the behavior of soccer agents. It turns out that humans very much rely on qualitative representations in their descriptions of tactics and strategies, particularly in the description of soccer tactics. We formulated several qualitative predicates which are essential for this application domain. The models we used are well-founded in the qualitative spatial reasoning community and are particularly useful as mathematical models for our needs. To be able to apply the formulated qualitative world model for soccer playing robots, one must be able to do some simple reasoning with the retrieved qualitative measures. Our approach to this is to exploit the fact that the qualitative world model is computed from quantitative data. With a hybrid representation of the world we are able to perform some simple reasoning like calculating the position of a robot which is moving far to the front-right direction. By this we are able to avoid spatial calculi which, in general, are computationally expensive. Very important to note is that with the presented qualitative world model we provide a state space abstraction for the robotic soccer domain. These abstractions allow to apply macro actions also for the complex soccer domain, as we have sketched at the end of this chapter. Of course, we are not the first to use qualitative predicates in general, and in the soccer domain in particular, and there is a lot of work in the related fields *spatial reasoning* and *state space abstractions*. In the following we are discussing some of the work on these fields.

Cohn and Hazarika give an overview of major qualitative spatial representation and reasoning techniques in their paper (Cohn and Hazarika 2001). They survey the main aspects of qualitative representations and they also consider methods for qualitative reasoning. Their survey covers ontological aspects and topological approaches as well as methods on distance, orientation, and shape. Besides, they mention possible applications for qualitative spatial reasoning which include geographical information systems, robot navigation, computer vision, engineering design, and many others. They evaluate the usefulness of different reasoning approaches with respect to their potential application areas. One of the most well known works on qualitative spatial reasoning is the region connection calculus (RCC).

In 1992 Randell, Cui, and Cohn presented a calculus for reasoning about regions and connections called RCC (Randell et al. 1992). The fundamental approach bases on extended spatial entities, that is, regions and the relations (also called connections) between them. The RCC allows for representing shape and structure of spatial objects. In contrast to systems from classic geometry the approach pursued in the RCC takes regions as primitives rather than points. The relation $C(x, y)$ states whether region x and region y are connected or not. The RCC is founded on two basic axioms on C, namely reflexivity $\forall x[C(x, x)]$ and symmetry $\forall x, y[C(x, y) \rightarrow C(y, x)]$, plus some axioms and definitions on the main spatial relations. With these axioms one can define predicates and functions that allow for the description of topological knowledge. All theorems and functions in the RCC are defined in first order logic. Thus, reasoning in the RCC can be performed by theorem proving. But, since first order logic is generally undecidable there is no effective way of reasoning with the complete version of the RCC in terms of computational complexity. Nevertheless, several efforts have been made to find subsets of the RCC which are more tractable.

For example, the RCC-8, a subset of the RCC, provides a set of eight basic relations between two regions x and y: disconnected $DC(x, y)$, part of $P(x, y)$, proper part of $PP(x, y)$, identical with $EQ(x, y)$, overlaps $O(x, y)$, partially overlaps $PO(x, y)$, externally connected $EC(x, y)$, tangential proper part $TPP(x, y)$, and non tangential proper part $NTPP(x, y)$. These eight relations are jointly exhaustive and pairwise disjoint. The RCC-8 has been well studied by Nebel and Renz (Renz and Nebel 1999). They proved that reasoning in the RCC-8 is NP-complete, in general, and they identify a maximal tractable subset called $\mathcal{H}_8$ that contains all basic relations. They observed that this language subset is sufficient to model important topological structures but they also noticed that it can be too weak for some other purposes. For a detailed account on this we refer to (Renz and Nebel 1999).

In the field of robotic soccer there exist some approaches to use qualitative notions to abstract from an infinite quantitative state space. Stolzenburg et al. (2002) investigate the use of qualitative velocities in the ROBOCUP Simulation league. Within their case study they compare four approaches to ball interception, namely a qualitative method, a numerical method, a strategy based on reinforcement learning, and a naive approach. Their focus is the qualitative representation of motion and they state that it was not covered that much in research on qualitative spatial reasoning. The experiments they present in (Stolzenburg et al. 2002) reveal that the qualitative method is less successful with ball interception than the numerical and the learning approach because the execution system still heavily depends on precise values to be successful. One has to remark that they conducted their experiments in a simulated environment. This means for instance that is easier to learn the artificial noise which is imposed by the simulation environment. It is doubtful if the same results apply in a real robotic environment

In recent work Beetz et al. (2005), overview the FIPM system, a real-time analysis tool for soccer games. Based on position data of the players and the ball they interpret common soccer concepts. In (Beetz et al. 2005) they report on first results drawn from data from the RoboCup simulation league. They use first-order interval temporal logic to represent events or situations. Their model consists of five layers comprising a motion, situation, action, and tactical layer. On the situation layer they identify concepts like *ScoringOpportunity*. With data-mining techniques they assess the conditions for such situations. On the action layer they distinguish between several kinds of models. The observation model for example classifies shots to belong to a dribbling or a pass, the predictive model use decision-tree learning to form rules for predicting the success rates of goal shots. They also provide, for example, information about the physical abilities of players based on the distances the player covers during a match, and also tactical patterns of a team can be derived. These information are especially useful for soccer coaches.

Miene et al. (Miene et al. 2003) report on successful experiments on detecting and predicting offside positions based on data also from the simulation league. They developed an algorithm for rule-based motion interpretation. The rules are given as background knowledge in first-order logic. The input data are first temporally segmented based on thresholds and monotonicity criteria. Then the segmented motion data are mapped into qualitative classes like *no motion*, or *slow*. They use logical representations to model game situations like a player being in an offside position. They are able to detect offside positions successfully and can also predict if a player risks to run into an offside trap. Important to note is that they do not regard static situations but analyse the motion data. This covers especially the dynamic aspects of soccer.

(Dylla et al. 2007) make use of the OPRA calculus to reason for way-rules for sail boats. In particular, they make use of the $OPRA_4$ calculus where two instances of oriented points yield the qualitative orientations. The $OPRA$ calculus allows for qualitative reasoning closed under converse and composition. The way rules are established using an ontology describing what it means that a vessel is approaching "from-the-star-board" or "from-the-harbor". Using an Ackermann kinematics for the boats collision-free qualitative courses are calculated.

In (Fraser et al. 2004) Fraser, Steinbauer, and Wotawa describe how to extract qualitative information from numerical world model data. Just like in our work they chose robotic soccer as the real world example domain to apply their qualitative data representation. Fraser et al. use a hybrid system similar to ours. It combines reactive low-level control with deliberative high-level decision making using classical AI approaches. Their planning approach uses STRIPS-like preconditions for every action. These preconditions are facts about perceived states of the world which are represented by Boolean predicates either being true or false such as the visibility of an object or the reachability of the ball. The mapping from quantitative values to their qualitative representation turns out to be not as easy as it might have looked like. Fraser et al. exemplary consider the inReach predicate which is grounded in the distance to the ball in order to determine whether it is reachable for the robot or not. Due to the unreliable perception of a robot, a simple threshold does not suffice as it leads to very unstable results. Hence, Fraser et al. propose the utilization of a so-called hysteresis function. Systems following a hysteresis function can be seen as systems that remember decisions taken previously. Once a predicate evaluates to true, it remains true unless a significant change in the world occurs. Steinbauer, Weber, and Wotawa discuss first results of the above work in (Steinbauer et al. 2005). They present results of several experiments conducted. The symbol grounding with hysteresis proposed in (Fraser et al. 2004) has proven useful to decrease the number of undesired changes in truth values of predicates to a minimum. This improvement in terms of stability in a robot's knowledge increases the performance of the decision making process. Although, some issues still remain unsolved. Firstly, the size of the hysteresis yet cannot be determined in general. Secondly, the conjunction of a large number of predicates using hysteresis has not been investigated sufficiently. Nonetheless, the concept of hysteresis may prove useful in our approach as well.

Chapter 8

Conclusion

Summary

In this thesis we proposed an approach to decision making of mobile robots or agents. The agent is acting in a dynamic environment with adversaries having opposing goals where it is forced to take decisions in real-time, i.e. in tenth of a second. We discussed several techniques how to come to rational decisions, which we as external observers would call intelligent. Intelligent decision making can be defined as taking decisions which the agent predicted to reach its goal. This presumes that the agent acts goal-directed and makes use of some form of deliberation. Deliberation is seen here mainly as predicting the effects of the own actions temporally, i.e. to reason which future world situation might evolve due to the actions of the agent. At the same time as the agent deliberates about future courses of actions it must be ensured that it stays reactive enough to cope with the real-time constraints posed by the environments we aim at. This means for one that the agent must be able to take decisions fast enough w.r.t. its own capabilities but also w.r.t. opposing agents which try to impede the own actions. Fast enough regarding its own capabilities means that the agent or robot has to keep up with the speed and decision cycles of its own actions. We gave the example of a dribbling robot. If the robot is dribbling with the ball it must take the next action fast not to lose the ball. Fast enough regarding opponents means that it cannot deliberate for a long time because a faster opponent will take advantage of this.

Therefore, one has to find a middle ground between deliberation and taking decisions in a reactive fashion. This means on the other hand to find the right level of abstraction for modeling the agent's or robot's action. We found our approach on the powerful situation calculus and the language GOLOG. With READYLOG we attain such a middle ground. It seamlessly integrates robot programming and deliberation. Programs can be partially specified and missing details are filled in by the READYLOG interpreter at run-time.

In READYLOG we follow two different approaches for deliberation. For one, we make use of a probabilistic projection mechanism. Programs which have a success probability associated can be projected into the future. In a future world situation one can query the likelihood that certain propositions hold. This can be used for decision making in the following way. Several alternatives are projected into the future, the one with the highest success probability is selected. We gave an example from the soccer domain. The agent planned a double pass with this technique. As we have pointed out, it is especially interesting to evaluate different models for the behaviors of opponents in the soccer scenario. Different behavior models of the opponents can be encoded as READYLOG programs with different probabilities. The agent now can find out under which model his own actions are most successful. If observations about the success of own actions under different opponent models are made one can reason about the appropriateness of the opponent model. This gives a means to address the problem of adopting to opponents behaviors (though we did not address it in this thesis).

The second approach we follow is that of decision-theoretic planning. Instead of querying whether certain propositions hold in some future situations, one assigns a utility to world states which are desirable for the agent. The background theory now finds a policy which maximizes the utility leading towards the desirable world states. Connected with this technique is a notion of uncertainty of ones own actions. This notion is indispensable for modeling dynamic real-time domains. Especially when dealing with embodied agents, i.e. robots, many different sources of uncertainty exist. These are the own perception and the own actuators of the robot, but there is also uncertainty about the actions of opponent robots. The action models, although being simplistic at the moment, have to account for these uncertainties. Again, there exists a trade-off between the complexity of the models and the computation time. Complex models lead to longer computation times for calculating behavior policies. We studied this with the "item pickup tasks" in UNREAL TOURNAMENT 2004.

Summarizing, READYLOG features probabilistic projections, decision-theoretic planning, but can also deal with continuous change. As a programming language it moreover melds useful features known from other robot programming languages, such as condition-bounded execution of action (wait-for) and external events. The framework of READYLOG as a high-level decision component can easily be integrated into state of the art software architectures for mobile robots. A layered software architecture is envisioned. The ability to directly connect READYLOG to a world model (on-line fluents), to settle commands to the low-level system (action register) while the high-level interpreter can concurrently follow other goals, and acquire information about the environment in the background (passive sensing) makes it very flexible and applicable for a wide range of robotics applications.

In the context of integrating decision theory into READYLOG, very important improvements are the execution monitoring facility of policies, the extension of the notion of stochastic actions (from an implementation point of view), and options. As one cannot always foresee how the world may evolve due to the inherent uncertainty of the application domain plans or policies might fail, i.e. the plan cannot provide a meaningful action for the world situation encountered. Then, it is important to quickly detect such situations to avoid that the agent is performing meaningless actions. With keeping track of the model assumptions made during planning, we are able to detect when a policy becomes invalid. At the moment, we discard the whole policy and start a re-planning process. It is a subject to future work to investigate other useful techniques such as plan repair to circumvent discarding the whole policy. The extension of modeling stochastic actions is probably only an issue of usability. We extended the notion of stochastic action outcomes in such a way that they can be modeled by programs instead of a single primitive action. The expressiveness, in general, is not improved but the implementor can more easily keep track of the outcome models. Further, we introduced the possibility to plan with macro actions in the decision-theoretic context. While our original approach had the drawback that it relied on explicit state space enumeration, the extension of using a DT plan library, where abstract policies are stored in, showed more flexibility. It allows to apply macro actions also for the complex soccer domain.

The issues above mirror the run-time aspects of READYLOG. It turns out that READYLOG with its expressiveness rooted in the situation calculus and GOLOG coming with a formal semantics is also well-suited as a behavior modeling and description language. It serves as a description language for our efforts to come to a qualitative description of soccer moves. Complex interactions in strategies can easily be modeled. Another nice feature of using READYLOG for describing complex behaviors is that we can run these descriptions on our robot without (too much) further modifications.

Finally, we set up a qualitative world model for soccer agents which allows for the formulation of strategies and high-level behaviors in a more natural, human-like manner. Finding models for the salient attributes of an application domain is one of the tasks to be achieved to establish a qualitative world model. Besides this has a qualitative world model the important property that it abstracts from quantitative world model representations. A qualitative predicate builds an equivalence class for a range of (infinitely many) quantitative assignments. This further allows to apply techniques like MDP macro actions also to continuous domains like soccer. This was shown for simulated soccer where macro actions could be formulated using the qualitative world model together with the aforementioned DT plan library.

With several example applications we showed the usefulness and applicability of READYLOG in simulations as with real robots. We also presented the robot system we built and its software components. In particular, we proposed a very general and reactive approach to collision avoidance and showed how one can localize a robot with proximity sensors in environments with sparse landmarks. The approach of representing “don’t care” regions in occupancy grid maps is of special value as it gives a means to hide regions in an occupancy map which seem not helpful for the localization task. Further, these regions can be used to stitch overlapping regions of several occupancy grid maps. This allows applications in large environments where the partial maps are stored in a memory-efficient way. The robot switches to another map when traversing such a “don’t care” region. Finally, we compared several state-of-the-art sensor fusion techniques with the problem of merging perceptions of the ball on a soccer field. A stable and good world model is important for making intelligent decisions and sensor fusions techniques help to improve the world model, especially when a team of robots can exchange their perceptions. The probabilistic methods can also be used to detect when a robot is likely to be dis-localized.

Future Work

As we have already stated before, are some of our results in a proof-of-concept state or slightly above, for instance, the DT plan library as discussed at the end of the previous chapter. In the current state we can formulate stochastic macro action very similar to those proposed in Chapter 4. The reason why this is possible is that the method seems to be general enough for this purpose and on the other hand, we yet only used small horizons to calculate the macro policy. This policy is executed when the macro action is called from the DT plan library. Currently, the calculation inside the macro consists of one-step DT plans. Here it would be interesting to find out if larger steps policies, like two-step or three-step policies are be beneficial. However, more investigations are needed to learn how well our proposed method scales.

Another very important and interesting field is the field of learning in the context of decision-theoretic planning and READYLOG. Looking at Reinforcement Learning methods it appeals that both, learning and planning, use the same formal background model. In the former, an explicit model is not given and has to be approximated, while the latter approach makes use of an explicit model (cf. also Chapter 3.1.2). The interesting question will be if it might be possible to use the information gathered from a run of Q-Learning to improve the action models on which the the planning procedure relies. As was done in RPLLEARN (Beetz et al. 2004) a future direction for READYLOG is to integrate learning methods directly into the framework. The advantage of such an integration is that the agent directly receives feedback from the environment about the performance of its action. A preliminary study how reinforcement learning techniques could be usefully integrated into the READYLOG framework, gave evidence that with the decision-theoretic extensions and our execution monitoring framework for policies it is easy to integrate RL methods by defining several simple $BestDo$ and $Trans$ predicates. The application we chose for a proof-of-concept implementation was a maze scenario. The model of the agent was the usual one, the intended action succeeds with a high probability and with the remaining probability mass the agent comes out in any adjacent position. The execution of the agent's action were now pathological. With the highest probability the agent came out in the opposite direction than was expected: if it intended to move to the right with probability 0.8 it came out left with this probability. We then compared the policies calculated from decision-theoretic planning and from our Q-learning READYLOG agent. The calculated policy never reached the goal position, while the Q-learner quickly found out which action it should take. The reason why the decision-theoretic agent could not proceed to the goal state was due to the precondition of its stochastic actions. In its start position in the lower left corner of the maze it has only the possibility to move to the right and upwards, the other two directions are blocked with walls. As it was not possible for the agent to take a go_left action (which with high probability lead the agent to the field to its right with our pathological execution system) due to its action preconditions he did not succeed. The learner instead had to try also the go_left action to explore its environment (one of the requirements of Q-learning) and thus were able to learn quickly that the actions had outcomes differently from expected.

Other directions focusing on learning the action models by different means are conceivable as well. A different approach to improve the effectiveness of READYLOG is to improve the action models used during policy generation of decision-theoretic planning. The way READYLOG chooses between action alternative is based on decision-theoretic planning. This means, that given a model of the outcomes of an action the decisions are made. Clearly, the models abstract from the world. In Chapter 5 we gave an example of an intercept-ball action. The failure case for this action, i.e. the outcome where the robot could not intercept the ball, was modeled such that it was assumed that the world did not change at all. While we argued in Chapter 5 that this behavior nevertheless exhibit the intended behavior, these models seem to be too simplistic. In his Diploma thesis Gester (2007) investigates models to learn the outcomes of actions. He uses methods from data mining to cluster the observation data in order to generate models which are as general as possible. The optimum would be to retrieve a logical description or a concise mathematical model of the outcomes of actions together with their probabilities which could directly transformed into the logical description in READYLOG. In his experiments he shows that for toy domains like the maze domain this works quite well, also the cluster methods applied generalize in a way that a fairly compact logical representation could be derived. For more realistic domains with a lot of noise and where the effects of an actions could only partially be observed (for instance, the soccer domain, where the effect and the termination of a ball passing action is hard to describe) the outcomes are less compact and generalization is weak. This opens a possible future research direction: what methods are suitable for learning the outcomes or how must the existing methods be improved

Finally, interesting applications for READYLOG can be found in service robotics applications. READYLOG should be extended in such a way that it is able to cope with the demands of modern service robotics applications. This means in first row to introduce extensions which allow a better integration of human-machine communication and dialog systems. The reasoning capabilities could be used to find a way to a target point as well as to tell the user that its commands are not clear. Being able to create an intelligently answering companion using projections and utility-based planning would have a major impact on the ways humans and robot communicate with each other. Today's service robotics competitions like RoboCup@Home show the abilities and the potential of mobile robots equipped with decent reasoning capabilities.

Index

Bibliography

Amiranashvili, V. (2007). *Robust Real-Time Localization and Mapping in Single and Multi-Robot Systems*. Ph. D. thesis, Knowledge-based Systems Groups, Computer Science Department, RWTH Aachen University.

Amiranashvili, V. and G. Lakemeyer (2005). Distributed multi-robot localization based on mutual path detection. In U. Furbach (Ed.), *KI 2005: Advances in Artificial Intelligence, Proceedings of the Twenty-Eighth Annual German Conference on Artificial Intelligence, (KI-05)*, Volume 3698 of *Lecture Notes in Computer Science*, pp. 279–290. Springer.

Apt, K. and M. Wallace (2006). *Constraint Logic Programming using Eclipse*. Cambridge University Press.

Arkin, R. (1986). Path planning for a vision-based mobile robot. In *Proceedings of the 1986 SPIE Conference on Mobile Robots (SPIE-86)*.

Arkin, R. (1987). Motor schema-based navigation for a mobile robot: An approach to programming by behavior. In *IEEE International Conference on Robotics and Automation (ICRA-87)*. IEEE Computer Society Press.

Arkin, R. (1989). Motor schema-based mobile robot navigation. *The International Journal of Robotics Research 8*(4), 92–112.

Arkin, R. (1998). *Behavior-Based Robotics*. MIT Press.

Aurenhammer, F. and R. Klein (2000). Voronoi diagrams. In J.-R. Sack and J. Urrutia (Eds.), *Handbook of Computational Geometry*, Chapter 5, pp. 201–290. North-Holland.

Bacchus, F., J. Halpern, and H. Levesque (1995). Reasoning about noisy sensors in the situation calculus. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1933–1940.

Bacchus, F., J. Halpern, and H. Levesque (1999). Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence 111*(1–2), 171–208.

Bahar, R., E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi (1993). Algebraic Decision Diagrams and Their Applications. In *Proceedings of the 1993 IEEE /ACM International Conference on CAD*, pp. 188–191. IEEE Computer Society Press.

Bar-Shalom, Y. and X. Li (1995). *Multitarget-Multisensor Tracking: Principles and Techniques*. YBS Publishing.

Baral, C. and T. Son (2000). Extending ConGolog to allow partial ordering. In N. R. Jennings and Y. Lespérance (Eds.), *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Volume 1757 of *Lecture Notes in Computer Science*. Springer.

Baral, C., N. Tran, and L.-C. Tuan (2002). Reasoning about actions in a probabilistic setting. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02) and Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-02)*, pp. 507–512.

Barto, A., S. Bradtke, and S. Singh (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence 72*(1), 81–138.

Bauckhage, C. and C. Thurau (2004). Exploiting the Fascination: Video Games in Machine Learning Research and Education. In *Proceedings of the 2nd International Workshop in Computer Game Design and Technology*, pp. 61–70. ACM.

Beetz, M. (1999). Structured reactive controllers. In O. Etzioni, J. P. Müller, and J. M. Bradshaw (Eds.), *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, pp. 228–235. ACM Press.

Beetz, M. (2000). Runtime plan adaptation in structured reactive controllers. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents-00)*, pp. 19–20. ACM Press.

Beetz, M. (2001). Structured reactive controllers. *Journal of Autonomous Agents and Multi-Agent Systems 2*(4), 25–55.

Beetz, M., B. Kirchlechner, and M. Lames (2005). Computerized real-time analysis of football games. *IEEE Pervasive Computing 4*(3), 33–39.

Beetz, M., A. Kirsch, and A. Müller (2004). RPLLEARN: Extending an autonomous robot control language to perform. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe (Eds.), *Proceedings of the Third International Joint Conference on Autonoumous Agents and Multi Agent Systems (AAMAS-04)*, pp. 1022–1029. ACM Press.

Beetz, M. and D. McDermott (1994). Improving robot plans during their execution. In K. Hammond (Ed.), *Proceedings of the Second International Conference on AI Planning Systems (AIPS-94)*, pp. 3–12. Morgan Kaufmann.

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

Bekey, G. (2005). *Autonomous Robots: From Biological Inspiration to Implementation and Control*. MIT Press.

Belker, T., M. Hammel, and J. Hertzberg (2003). Learning to optimize mobile robot navigation based on HTN plans. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, (ICRA-03)*, pp. 4136–4141. IEEE Computer Society Press.

Belleghem, K. V., M. Denecker, and D. D. Schreye (1997). On the relation between situation calculus and event calculus. *Journal of Logic Programming 31*(1-3), 3–37.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princton, NJ.

Bererton, C. (2004). State Estimation for Game AI Using Particle Filters. In *AAAI-04 Workshop on Challenges in Game AI*. AAAI-04.

Bertsekas, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall.

Bertsekas, D. and J. Tsitsiklis (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bjärland, M. (1999). Recovering from modeling faults in Golog. In IJCAI-99 (Ed.), *IJCAI-99 Workshop: Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World*.

Boddy, M. (1991). *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments*. Ph. D. thesis, Department of Computer Science at Brown University, Birmingham, UK.

Böhnstedt, L. (2007). Macro-actions for highly dynamic domains in readylog. Diploma thesis, Knowledge-based Systems Group, Computer Science Department, RWTH Aachen University.

Böhnstedt, L., A. Ferrein, and G. Lakemeyer (2007). Options in readylog reloaded – generating decision-theoretic plan libraries in golog. In *Proceeding of the 30th German National Conference on Artificial Intelligence*. to appear.

Bonarini, A., M. Matteucci, and M. Restelli (2001). Anchoring: do we need new solutions to an old problem or do we have old solutions for a new problem? In *Proceedings of the AAAI Fall Symposium on Achoring Symbols to Sensor Data in Single and Multiple Robot Systems*. AAAI Press.

Bonasso, R., R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artifical Intelligence 9*(2-3), 237–256.

Borenstein, J. and Y. Koren (1991). The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation 3*(7), 278–288.

Boutilier, C., T. Dean, and S. Hanks (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research 11*, 1–94.

Boutilier, C., R. Dearden, and M. Goldszmidt (1995). Exploiting structure in policy construction. In C. Mellish (Ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1104–1111. Morgan Kaufmann.

Boutilier, C., R. Dearden, and M. Goldszmidt (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence 121*(1-2), 49–107.

Boutilier, C. and D. Poole (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI-96)*, pp. 1168–1175. AAAI Press / The MIT Press.

Boutilier, C., R. Reiter, and B. Price (2001). Symbolic dynamic programming for first-order MDPs. In B. Nebel (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, (IJCAI-01)*, pp. 690–700. Morgan Kaufmann.

Boutilier, C., R. Reiter, M. Soutchanski, and S. Thrun (2000). Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pp. 355–362. AAAI Press.

Bratman, M. (1987). *Intentions, Plans, and Practical Reason*. Harvard University Press.

Bredenfeld, A., A. Jacoff, I. Noda, and Y. Takahashi (Eds.) (2006). *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*. Springer.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*(1), 14–23.

Brooks, R. and S. Iyengar (1998). *Multi-sensor fusion: Fundamentals and Applications with Software*. Prentice Hall.

Brooks, R. A. (1991). Intelligence without representation. *Artifical Intelligence 47*(1-3), 139–159.

Buehler, M., K. Iagnemma, and S. Singh (Eds.) (2007). *The 2005 DARPA Grand Challenge – The Great Robot Race*, Volume 36 of *Springer Tracts in Advanced Robotics*. Sptringer Verlag.

Burgard, W., A. Cremers, D. Fox, G. Lakemeyer, D. Hähnel, D. Schulz, W. Steiner, and S. Thrun (1998). The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) and Tenth Innovative Applications of Artificial Intelligence Conference (IAAI-98)*. AAAI Press.

Burgard, W., M. Moors, C. Stachniss, and F. Schneider (2005). Coordinated multi-robot exploration. *IEEE Transactions on Robotics 21*(3), 376–378.

Buro, M. (2003). Real-time strategy games: A new AI research challenge. In G. Gottlob and T. Walsh (Eds.), *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pp. 1534–1535. Morgan Kaufmann.

Calmes, L., A. Ferrein, G. Lakemeyer, and H. Wagner (2006). Von Schleiereulen und fuŝsballspielenden Roboter. *RWTH Themen* (1), 30–33. (in German).

Calmes, L., H. Wagner, S. Schiffer, and G. Lakemeyer (2007). Combining sound localization and laser-based object recognition. In *Proceedings of the AAAi Spring Symposium 2007*.

Cassandra, A., L. Kaelbling, and M. Littman (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Volume 2, pp. 1023–1028. AAAI Press/MIT Press.

Chen, T. and K. Chung (2001). An efficient randomized algorithm for detecting circles. *Computer Vision and Image Understanding 83*(2), 172–191.

Chung, H., L. Ojeda, and J. Borenstein (2001). Sensor fusion for mobile robot dead-reckoning with a precision-calibrated fiber optic gyroscope. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-01)*. IEEE Computer Society Press.

Claßen, J., P. Eyerich, G. Lakemeyer, and B. Nebel (2007). Towards an integration of golog and planning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*. AAAI Press.

Claßen, J. and G. Lakemeyer (2006). Foundations for knowledge-based programs using ES. In *Proceedings of the Tenth Conference on Principles of Knowledge Representation and Reasoning (KR-06)*. AAAI Press.

Clementini, E., P. di Felice, and D. Hernàndez (1997). Qualitative representation of positional information. *Artificial Intelligence 95*(2), 317–356.

Cohn, A. G. and S. M. Hazarika (2001). Qualitative Spatial Representation and Reasoning: An Overview. *Fundamenta Informaticae 46*(1-2), 1–29.

CoSy (2007). http://www.cognitivesystems.org/abstract.asp. last visited in January.

Davis, E. (1994). Knowledge preconditions for plans. *Journal of Logic and Computation 4*(5), 721–766.

De Giacomo, G., Y. Lesperance, and H. Levesque (1997). Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In M. Pollack (Ed.), *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann.

De Giacomo, G., Y. Lespérance, H. Levesque, and S. Sardiña (2002). On the semantics of deliberation in IndiGolog – from theory to implementation. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams (Eds.), *Proceedings of Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, pp. 603–614. Morgan Kaufmann.

De Giacomo, G., Y. Lesperance, and H. J. Levesque (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence 121*(1-2), 109–169.

De Giacomo, G. and H. Levesque (1999). An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri (Eds.), *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pp. 86–102. Springer.

De Giacomo, G., H. Levesque, and S. Sardiña (2001). Incremental execution of guarded theories. *Computational Logic 2*(4), 495–525.

De Giacomo, G., Y. Lsperance, and H. Levesque (2000). ConGolog, A concurrent programming language based on situation calculus. *Artificial Intelligence 121*(1–2), 109–169.

De Giacomo, G., R. Reiter, and M. Soutchanski (1998). Execution monitoring of high-level robot programs. In A. Cohn, L. Schubert, and S. Shapiro (Eds.), *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pp. 453–465. Morgan Kaufmann.

Dean, T. and R. Givan (1997). Model minimization in markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97) and Ninth Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pp. 106–111. AAAI Press / The MIT Press.

Dean, T. and K. Kanazawa (1990). A model for reasoning about persistence and causation. *Computational Intelligence 5*(3), 142–150.

Dellaert, F., D. Fox, W. Burgard, and S. Thrun (1999). Monte Carlo localization for mobile robots. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation, (ICRA-99)*. IEEE Computer Society Press.

DeLoura, M., D. Treglia, A. Kirmse, K. Pallister, and M. Dickheiser (Eds.) (2000 – 2006). *Game Programming Gems*, Volume 1–6. Charles River Media.

Dietl, M., J.-S. Gutmann, and B. Nebel (2001). Cooperative sensing in dynamic environments. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-01)*. IEEE Computer Society Press.

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

Dimarogonas, D. and K. Kyrikopoulos (2005). A Feedback Stabilization and Collision Avoidance Scheme for Multiple Independent Nonholonomic Non-point Agents. In *Proceedings of the 2005 International Symposium on Intelligent Control & 13th Mediterranean Conference on Control and Automation*.

dmoz.org/Computers/Robotics (2007). Web ressources in the web. http://dmoz.org/ Computers/ Robotics/. (last visited January).

Doherty, P. (2005). Knowledge representation and unmanned aerial vehicles. In A. Skowron, R. Agrawal, M. Luck, T. Yamaguchi, P. Morizet-Mahoudeaux, J. Liu, and N. Zhong (Eds.), *Proceedings of the 2005 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2005)*, pp. 9–16. IEEE Computer Society Press.

Doherty, P., G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund (2000). The WITAS unmanned aerial vehicle project. In W. Horn (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*. IOS Press.

Doherty, P., J. Gustafsson, L. Karlsson, and J. Kvarnstrom (1998). TAL: Temporal action logics – language specification and tutorial. *Linkoping Electronic Articles in Computer and Information Science 15*(3), 273–306.

Doherty, P., P. Haslum, F. Heintz, T. Merz, T. Persson, and B. Wingman (2004). A distributed architecture for intelligent unmanned aerial vehicle experimentation. In *Proceedings of the Seventh International Symposium on Distributed Autonomous Robotic Systems*.

Dylla, F., A. Ferrein, and G. Lakemeyer (2003a). AllemaniACs 2004 team description.

Dylla, F., A. Ferrein, and G. Lakemeyer (2003b). Specifying multirobot coordination in ICPGolog – from simulation towards real robots. In *Proceedings of the Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modeling, planning, learning, and communicating*. IJCAI-03.

Dylla, F., A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, S. Schiffer, F. Stolzenburg, U. Visser, and T. Wagner (2007). Approaching a formal soccer theory from behaviour specifications in robotic soccer. In P. Dabnicki and A. Baca (Eds.), *Computer in Sports*, pp. . WIT Press. accepted for publication.

Dylla, F., A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, F. Stolzenburg, U. Visser, and T. Wagner (2005). Towards a League-Independent Qualitative Soccer Theory for Robocup. In D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor (Eds.), *RoboCup 2004: Robot Soccer World Cup VIII*, Volume 3276 of *Lecture Notes in Computer Science*. Springer.

Dylla, F., L. Frommberger, J. Wallgrün, D. Wolter, S. Wölfl, and B. Nebel (2007). Sailaway: Formalizing navigation rules. *Proceedings of the AISB'07 Artificial and Ambient Intelligence Symposium on Spatial Reasoning and Communication*.

Epic Games Inc. (2007, last visited in MayFebruary). http://www.unrealtournament.com/.

Erol, K., J. Hendler, and D. Nau (1994a). HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Volume 2, pp. 1123–1128. AAAI Press/MIT Press.

Erol, K., J. Hendler, and D. Nau (1994b). Semantics for hierarchical task network planning. Technical Report CS-TR-32391, UMIACS-TR-94-31, ISR-TR-95-9, Computer Science Department, University of Maryland.

Erol, K., J. Hendler, and D. Nau (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence 18*(1), 69–93.

Farinelli, A., A. Finzi, and T. Lukasiewicz (2007). Team programming in golog under partial observability. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*.

Feng, Z. and E. Hansen (2002). Symbolic heuristic search for factored markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02) and Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-02)*. AAAI Press.

Ferrein, A. (2004a). Planned economy. *Linux Magazin* (7), 50–53. (in German).

Ferrein, A. (2004b). Specifying soccer moves with golog. In *Proceedings of the 5th Conference dvs-Section Computer Science in Sport*.

Ferrein, A., C. Fritz, and G. Lakemeyer (2003). Extending DTGOLOG with options. In G. Gottlob and T. Walsh (Eds.), *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann.

Ferrein, A., C. Fritz, and G. Lakemeyer (2004). On-line decision-theoretic golog for unpredictable domains. In S. Biundo, T. W. Frühwirth, and G. Palm (Eds.), *KI 2004: Advances in Artificial Intelligence, Proceedings of the Twenty-Seventh Annual German Conference on Artificial Intelligence, (KI-04)*, Volume 3238 of *Lecture Notes in Computer Science*, pp. 322–336. Springer.

Ferrein, A., C. Fritz, and G. Lakemeyer (2005a). AllemaniACs 2004 team description. In D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor (Eds.), *RoboCup 2004: Robot Soccer World Cup VIII*, Volume 3276 of *Lecture Notes in Computer Science*. Springer.

Ferrein, A., C. Fritz, and G. Lakemeyer (2005b). Using golog for deliberation and team coordination in robotic soccer. *KI 19*(1), 24–30.

Ferrein, A., C. Fritz, and G. Lakemeyer (2006). AllemaniACs 2005 team description. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi (Eds.), *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*. Springer.

Ferrein, A., L. Hermanns, and G. Lakemeyer (2006). Comparing sensor fusion techniques for ball position estimation. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi (Eds.), *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*, pp. 154–165. Springer.

Ferrein, A. and G. Lakemeyer (2005). Wie roboter die welt sehen. In A. Beyer and M. Lohoff (Eds.), *Bild und Erkenntnis – Formen und Funktion des Bildes in Wissenschaft und Technik*. Deutscher Kunstverlag und RWTH Aachen. (in German).

Ferrein, A. and G. Lakemeyer (2006). Fuballroboter – Wissenschaft, die auch Spa macht. *RWTH Themen* (2), 36–39. (in German).

Ferrein, A., G. Lakemeyer, and S. Schiffer (2007a). AllemaniACs 2006 team description. In G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi (Eds.), *RoboCup 2006: Robot Soccer WorldCup X*, Lecture Notes in Computer Science. Springer.

Ferrein, A., G. Lakemeyer, and S. Schiffer (2007b). AllemaniACs@home 2006 team description. In G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi (Eds.), *RoboCup 2006: Robot Soccer WorldCup X*, Lecture Notes in Computer Science. Springer.

Fichtner, M., A. Großmann, and M. Thielscher (2003). Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae 57*(2–4), 371–392.

Fikes, R. and N. Nilson (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 189–208.

Finzi, A. and T. Lukasiewicz (2004). Game-theoretic agent programming in Golog. In R. L. de Mántaras and L. Saitta (Eds.), *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI-04), including Prestigious Applicants of Intelligent Systems, (PAIS-04)*, pp. 23–27. IOS Press.

Finzi, A. and T. Lukasiewicz (2005). Game-theoretic Golog under partial observability. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. Singh, and M. Wooldridge (Eds.), *Proceedings of 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-05)*, pp. 1301–1302. ACM Press.

Finzi, A. and F. Pirri (2004). Flexible interval planning in concurrent temporal golog. In *The Fourth international Cognitive Robotics Workshop*, pp. 102–107.

Firby, R. (1987). An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 202–206. AAAI Press.

Firby, R. (1994). Task networks for controlling continuous processes. In K. Hammond (Ed.), *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*. AAAI Press.

Firby, R. J. (1996). Modularity issues in reactive planning. In B. Drabble (Ed.), *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 78–85. AAAI Press.

Firby, R. J., R. Kahn, P. Prokopowicz, and M. Swain (1995). An architecture for vision and action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 72–81.

Forbus, K., J. Mahoney, and K. Dill (2002). How Qualitative Spatial Reasoning Can Improve Strategy Game AIs. *IEEE Intelligent Systems 17*(4), 25–30.

Fox, D. (2001). KLD-sampling: Adaptive particle filters. In *Advances in Neural Information Processing Systems 14*. MIT Press.

Fox, D., W. Burgard, and S. Thrun (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine 4*(1), 23–33.

Fox, D., W. Burgard, and S. Thrun (1999). Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research 11*, 391–427.

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

Fraser, G., G. Steinbauer, and F. Wotawa (2004). Application of qualitative reasoning to robotic soccer. In *Proceedings of the Eighteenth International Workshop on Qualitative Reasoning*, pp. 173–178.

Freksa, C. (1992). Using orientation information for qualitative spatial reasoning. In A. Frank, I. Campari, and U. Formentini (Eds.), *Proceedings of the International Conference (GIS-92) - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, Volume 639 of *Lecture Notes in Computer Science*, pp. 162–178. Springer.

Freksa, C. and K. Zimmermann (1992). On the utilization of spatial structures for cognitively plausible and efficient reasoning. In *IEEE International Conference on Systems Man and Cybernetics*, pp. 261–266. IEEE Computer Society Press.

Fritz, C. (2003). Integrating decision-theoretic planning and programming for robot control in highly dynamic domains. Diploma thesis, Knowledge-based Systems Group, Computer Science V, RWTH Aachen, Aachen, Germany.

Fritz, C. (2004). No toys. *Linux Magazin* (7), 46–49. (in German).

Fritz, C. and S. McIlraith (2005). Compiling qualitative preferences into decision-theoretic Golog programs. In *The Sixth Workshop on Nonmonotonic Reasoning, Action, and Change, August 1 (NRAC-05)*. IJCAI-05.

Funge, J. (1998). *Making Them Behave: Cognitive Models for Computer Animation*. Ph. D. thesis, University of Toronto, Toronto, Canada.

Funge, J. (2000). Cognitive modeling for games and animation. *Commununications of the ACM 43*(7), 40–48.

Gabaldon, A. (2006). Formalizing complex task libraries in golog. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso (Eds.), *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06), Including Prestigious Applications of Intelligent Systems (PAIS-06)*, pp. 755–756. IOS Press.

Gabaldon, A. and G. Lakemeyer (2007). Esp: A logic of only-knowing, noisy sensing and acting. In *Twenty-Second Conference on Artificial Intelligence (AAAI-07)*. AAAI Press. to appear.

Galliers, J. (1988). *A Theoretical Framework for Computer Models of Cooerative Dialogue, Acknowleding Multi-Agent Conflict*. Ph. D. thesis, Open University, UK.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In W. Swartout (Ed.), *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 809–815. AAAI Press.

Gat, E. (1998). On three layer architectures. In D. Kortenkamp, R. Bonasso, and R. Murphy (Eds.), *Artificial Intelligence and Mobile Robots*, Chapter 8, pp. 195–210. MIT/AAAI Press.

Geffner, H. and B. Bonet (1998). High-level planning and control with incomplete information using POMDPs.

Gelfond, M. and V. Lifschitz (1993). Representing action and change by logic programs. *Journal of Logic Programming 17*(2/3&4), 301–321.

Gelfond, M. and V. Lifschitz (1998). Action languages. *Electronic Transactions on Artificial Intelligence 2*, 193–210.

Genesereth, M., N. Love, and B. Pell (2005). General game playing: Overview of the AAAI competition. *AI Magazine 26*(2), 62–72.

Georgeff, M. and A. Lansky (1986). Procedural knowledge. *Proceedings of the IEEE, Special Issue on Knowledge Representation 74*(10), 1383–1398.

Gester, C. (2007). Observing models of stochastic actions in the Readylog framework. Diploma thesis, Knowledge-based Systems Group, Computer Science Department, RWTH Aachen University.

GGP (2006). General game playing project competition. http://games.stanford.edu/.

Ginsberg, M. and D. Smith (1988). Reasoning about action II: The qualification problem. *Artificial Intelligence 35*(3), 311–342.

Giunchiglia, E., J. Lee, V. Lifschitz, N. McCain, and H. Turner (2004). Nonmonotonic causal theories. *Artificial Intelligence 153*(1-2), 49–104.

Goodwin, R. (1995). Formalizing properties of agents. *Journal of Logic Computation 6*(5), 763–781.

Gottfried, B. (2005). Collision avoidance with bipartite arrangements. In *Proceedings of the 2005 ACM workshop on Research in knowledge representation for autonomous systems*, pp. 9–16. ACM Press.

Green, C. (1969). Application of theorem proving to problem solving. In D. Walker and L. Norton (Eds.), *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*. William Kaufmann.

Grosskreutz, H. (2000). Probabilistic projection and belief update in the pgolog framework. In *The Second International Cognitive Robotics Workshop, (CogRob-00)*, pp. pages 34–41. ECAI-00.

Grosskreutz, H. (2002). *Towards More Realistic Logic-Based Robot Controllers in the GOLOG Framework*. Ph. D. thesis, Knowledge-based Systems Group, Computer Science V, RWTH Aachen.

Grosskreutz, H. and G. Lakemeyer (2000a). cc-Golog: Towards more realistic logic-based robot controllers. In AAAI Press (Ed.), *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*. AAI Press/ The MIT Press.

Grosskreutz, H. and G. Lakemeyer (2000b). Turning high-level plans into robot programs in uncertain domains. In W. Horn (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*. IOS Press.

Grosskreutz, H. and G. Lakemeyer (2001). On-line execution of cc-Golog plans. In B. Nebel (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, (IJCAI-01)*. Morgan Kaufmann.

Grosskreutz, H. and G. Lakemeyer (2003). ccgolog – A logical language dealing with continuous change. *Logic Journal of the IGPL 11*(2), 179–221.

Großmann, A., S. Hölldobler, and O. Skvortsova (2002). Symbolic dynamic programming with the Fluent Calculus. In *Proceedings IASTED Artificial and Computational Intelligence*, pp. 378–383.

Gu, Y. (2003). Macro-actions in the situation calculus. In *Proceedings of IJCAI-03 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC-03)*. IJCAI-03.

Gutmann, J.-S. (2002). Markov-Kalman localization for mobile robots. In *Proceedings of the Sixteenth International Conference on Pattern Recognition (ICPR-02)*, Volume 2.

Gutmann, J.-S. and D. Fox (2002). An experimental comparison of localization methods continued. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-02)*. IEEE Computer Society Press.

Gutmann, J.-S., W. Hatzack, I. Herrmann, B. Nebel, F. Rittinger, A. Topor, T. Weigel, and B. Welsch (1999). The CS freiburg robotic soccer team: Reliable self-localization, multirobot sensor integration, and basic soccer skills. In M. Asada and H. Kitano (Eds.), *RoboCup-98: Robot Soccer World Cup II*, Volume 1604 of *Lecture Notes in Computer Science*, pp. 93–108. Springer.

Hähnel, D., W. Burgard, and G. Lakemeyer (1998). GOLEX - bridging the gap between logic Golog and a real robot. In O. Herzog and A. Günter (Eds.), *KI-98: Advances in Artificial Intelligence, Proceedings of the Twenty-Second Annual German Conference on Artificial Intelligence*, Volume 1504 of *Lecture Notes in Computer Science*, pp. 165–176. Springer.

Hall, D. L. (2004). *Mathematical Techniques in Multisensor Data Fusion* (2nd ed.). Artech House, Inc.

Hauskrecht, M., N. Meuleau, L. Kaelbling, T. Dean, and C. Boutilier (1998). Hierarchical solutions of MDPs using macro-actions. In C. Cooper and S. Moral (Eds.), *Proceedings of the Fourteenth Conference on Uncertainty in AI (UAI-98)*. Morgan Kaufmann.

Hermanns, L. (2004). Fusing uncertain world information of cooperating robots into a global world model. Diploma thesis, Knowledge-based Systems Group, Computer Science V, RWTH Aachen, Aachen, Germany, (in German).

Hernández, D. (1991). Relative representation of spatial knowledge: The 2-D case. In D. Mark and A. Frank (Eds.), *Cognitive and Linguistic Aspects of Geographic Space*, pp. 373–385. Kluwer.

Hernández, D., E. Clementini, and P. di Felice (1995). Qualitative distances. In A. U. Frank and W. Kuhn (Eds.), *Spatial Information Theory: A Theoretical Basis for GIS, International Conference COSIT '95, Semmering, Austria, September 21-23, 1995, Proceedings*, Volume 988 of *Lecture Notes in Computer Science*, pp. 45–57. Springer.

Hindriks, K., F. de Boer, W. van der Hoek, and J.-J. Meyer (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems 4*(2), 357–401.

Hindriks, K., Y. Léesperance, and H. Levesque (2000). An embedding of ConGolog in 3APL. In W. Horn (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*, pp. 558–562. IOS Press.

Hoey, J., R. St-Aubin, A. Hu, and C. Boutilier (1999). SPUDD: Stochastic planning using decision diagrams. In K. Laskey and H. Prade (Eds.), *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pp. 279–288. Morgan Kaufmann.

Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Hu, Y. (2006). A declarative semantics of a subset of PDDL with time and concurrency. Master's thesis, Knowledge-Based Systems Group, Computer Science Department, RWTH Aachen University.

Ilghami, O., H. Muñoz-Avila, D. Nau, and D. Aha (2005). Learning approximate preconditions for methods in hierarchical plans. In L. D. Raedt and S. Wrobel (Eds.), *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML-05)*, pp. 337–344. ACM Press.

Ilghami, O., D. S. Nau, H. Muñoz-Avila, and D. W. Aha (2002). CameL: Learning method preconditions for HTN planning. In M. Ghallab, J. Hertzberg, and P. Traverso (Eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, pp. 131–142. AAAI Press.

Iwan, G. (2002). History-based diagnosis templates in the framework of the situation calculus. *AI Commununications 15*(1), 31–45.

Jacobs, S. (2005). Applying readylog to agent programming in interactive computer games. Diploma thesis, Knowledge-based Systems Group, Computer Science Department, RWTH Aachen University.

Jacobs, S., A. Ferrein, and G. Lakemeyer (2005a). Controlling unreal tournament 2004 bots with the logic-based action language golog. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-05)*. Demonstration.

Jacobs, S., A. Ferrein, and G. Lakemeyer (2005b). Unreal Golog bots. In *IJCAI-05 WS on Reasoning, Representation, and Learning in Computer Games*.

Jansen, N. (2002). A framework for developing deliberative components in uncertain, highly dynamic domains with real-time constraints. Diploma thesis, Knowledge-based Systems Group, Computer Science V, RWTH Aachen,Aachen, Germany (in German).

Jenkin, M., Y. Lespérance, H. Levesque, F. Lin, J. Lloyd, D. Marcu, R. Reiter, R. Scherl, and K. Tam (1997). A logical approach to portable high-level robot programming. In *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence (AI-97)*.

Jensen, R. and M. Veloso (1998). Interleaving deliberative and reactive planning in dynamic multi-agent domains. In *Proceedings of the AAAI Fall Symposium on on Integrated Planning for Autonomous Agent Architectures*. AAAI Press.

Kaelbling, L. P., M. L. Littman, and A. P. Moore (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4*, 237–285.

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering 82*(1), 34–45.

Kaminka, G., M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. Marshall, A. Scholder, and S. Tejada (2002). Game bots: A flexible test bed for multiagent research. *Communications of the ACM 45*(2), 43–45.

Kelleher, J. and G.-J. Kruijff (2006). Incremental generation of spatial referring expressions in situated dialog. In C. Cardie and P. Isabelle (Eds.), *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL-06)*. The Association for Computer Linguistics.

Kelleher, J., G.-J. Kruijff, and F. Costello (2006). Proximity in context: An empirically grounded computational model of proximity for processing topological spatial expressions. In C. Cardie and P. Isabelle (Eds.), *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL-06)*. The Association for Computer Linguistics.

Kitano, H., M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa (1997). Robocup: The robot world cup initiative. In L. Johnson (Ed.), *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*, pp. 340–347. ACM Press.

Kok, J. R. and N. A. Vlassis (2006). Using the max-plus algorithm for multiagent decision making in coordination graphs. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi (Eds.), *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*, pp. 1–12. Springer.

Konolige, K. (1997). COLBERT: A language for reactive control in sapphira. In *KI - Küunstliche Intelligenz*, pp. 31–52.

Konolige, K., K. Myers, E. Ruspini, and A. Saffiotti (1997). The Saphira architecture: A design for autonomy. *Journal of Experimental & Theoretical Artificial Intelligence 9*(1), 215–235.

Kortenkamp, D., R. Bonasso, and R. Murphy (Eds.) (1998). *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press.

Kowalski, R. (1992). Database updates in the event calculus. *Journal of Logic Programming 12*(1-2), 121–146.

Kowalski, R. and M. Sergot (1986). A logic-based calculus of events. *New Generation Computing 4*, 67–95.

Kruijff, G.-J., J. Kelleher, G. Berginc, and A. Leonardis (2006). Structural descriptions in human-assisted robot visual learning. In *Proceeding of the First ACM SIGCHI/SIGART Conference on Human-robot Interaction (HRI-06)*, pp. 343–344. ACM Press.

Kushmerick, N., S. Hanks, and D. S. Weld (1995). An algorithm for probabilistic planning. *Artificial Intelligence 76*(1-2), 239–286.

Kvarnström, J., P. Doherty, and P. Haslum (2000). Extending TALplanner with concurrency and resources. In W. Horn (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*, pp. 501–505. IOS Press.

Laird, J., M. Assanie, B. Bachelor, N. Benninghoff, S. Enam, B. Jones, A. Kerfoot, C. Lauver, B. Magerko, J. Sheiman, D. Stokes, and S. Wallace (2002). A test bed for developing intelligent synthetic characters. In *Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment 2002*. AAAI Press.

Lakemeyer, G. (1996). Only knowing in the situation calculus. In C. Aiello, J. Doyle, and S. Shapiro (Eds.), *Proceedings of Fifth International Conference in Principles of Knowledge Representation and Reasoning (KR-96)*, pp. 14–25. Morgan Kaufmann.

Lakemeyer, G. (1999). On sensing and off-line interpreting in GOLOG. In H. Levesque and F. Pirri (Eds.), *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pp. 173–187. Springer.

Lakemeyer, G. and H. J. Levesque (1998). AOL: A logic of acting, sensing, knowing, and only knowing. In A. Cohn, L. Schubert, and S. Shapiro (Eds.), *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pp. 316–329. Morgan Kaufmann.

Lakemeyer, G. and H. J. Levesque (2004). Situations, si! situation terms, no! In D. Dubois, C. Welty, and M.-A. Williams (Eds.), *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR-04)*. AAAI Press.

Lakemeyer, G. and H. J. Levesque (2005). Semantics for a useful fragment of the situation calculus. In L. P. Kaelbling and A. Saffiotti (Eds.), *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pp. 490–496. Professional Book Center.

Lakemeyer, G., E. Sklar, D. Sorrenti, and T. Takahashi (Eds.) (2007). *RoboCup 2006: Robot Soccer WorldCup X*, Lecture Notes in Computer Science. Springer.

Lauer, M. and M. A. Riedmiller (2000). An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In P. Langley (Ed.), *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-00)*, pp. 535–542. Morgan Kaufmann.

Levesque, H. (1996). What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI-96)*. AAAI Press / The MIT Press.

Levesque, H. and M. Pagnucco (2000). Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the Second International Cognitive Robotics Workshop (CogRob-00)*. ECAI-00.

Levesque, H., F. Pirri, and R. Reiter (1998). Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science 2*(18), 159–178. http://www.ep.lui.se/ea/cis/1998/018.

Levesque, H., R. Reiter, Y. Lesperance, F. Lin, and R. Scherl (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31*(1-3), 59–83.

Levesque, H. J. (1990). All I know: A study in autoepistemic logic. *Artificial Intelligence 42*(2-3), 263–309.

Levesque, H. J. and G. Lakemeyer (2001). *The Logic of Knowledge Bases*. MIT Press.

Lewis, R. (1999). Cognitive modeling, symbolic. In *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press.

Lifschitz, V. (1994). Circumscription. In D. Gabbay, C. Hogger, and J. Robinson (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pp. 298–352. Oxford University Press.

Lin, F. and R. Reiter (1994). State constraints revisited. *Journal of Logic Computing. 4*(5), 655–678.

Lin, F. and R. Reiter (1995). How to progress a database II: The STRIPS connection. In *IJCAI*, pp. 2001–2009.

Lin, F. and R. Reiter (1997). How to progress a database. *Artificial Intelligence 92*(1-2), 131–167.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, New Brunswick, NJ, pp. 157–163. Morgan Kaufmann.

Lötzsch, M., J. Bach, H.-D. Burkhard, and M. Jngel (2004). Designing agent behavior with the extensible agent behavior specification language XABSL. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida (Eds.), *RoboCup 2003: Robot Soccer World Cup VII*, Volume 3020 of *Lecture Notes in Computer Science*. Springer.

Lovejoy, W. (1991). Computationally feasible bounds for partially observable markov processes. *Operations Research 39*, 162–175.

Lucchesi, M. (2001). *Coaching the 3-4-1-2 and 4-2-3-1*. Reedswain Publishing.

Maes, P. (1989). How to do the right thing. *Connection Science Journal 1*(3), 291–323. Special Issue on Hybrid Systems.

Magerko, B., J. Laird, M. Assanie, A. Kerfoot, and D. Stokes (2004). AI characters and directors for interactive computer games. In D. McGuinness and G. Ferguson (Eds.), *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04) and Sixteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-04)*. AAAI Press / The MIT Press.

Martin, Y. (2003). The concurrent, continuous FLUX. In G. Gottlob and T. Walsh (Eds.), *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann.

Maybeck, P. (1979). *Stochastic Models, Estimation and Control*, Volume 1. Academic Press.

Maybeck, P. (1990). The Kalman filter: an introduction to concepts. In L. Cox and J. Wilfong (Eds.), *Autonomous Robot Vehicles*. Springer.

McAllester, D. and D. Rosenblitt (1991). Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, Volume 2, pp. 634–639. AAAI Press / The MIT Press.

McCarthy, J. (1963). Situations, actions and causal laws. Technical report, Stanford University.

McCarthy, J. (1980). Circumscription – A from of non-monotonic reasoning. *Artifical Intelligence 1–2*(13), 27–39.

McCarthy, J. (1985). Formalization of STRIPS in SITUATION CALCULUS. http://www-formal.stanford.edu/jmc/strips/strips.html.

McCarthy, J. and P. Hayes (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, 463–502.

McDermott, D. (1991). A reactive plan language. Technical Report YALEU/DCS-RR-864, Yale University, Department of Computer Science.

McDermott, D. (1992). Transformational planning of reactive behavior. Technical Report YALEU/DCS/RR-941, Yale University, Department of Computer Science.

McDermott, D. (1994). An algorithm for probabilistic, totally-ordered temporal projection. Technical Report YALEU/DCS/RR-1014, Yale University, Department of Computer Science.

McIlraith, S. and T. Son (2002). Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams (Eds.), *Proceedings of Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*. Morgan Kaufmann.

Meier, D., C. Stachniss, and W. Burgard (2006). Cooperative exploration with multiple robots using low bandwidth communication. In J. Beyerer, F. P. León, and K.-D. Sommer (Eds.), *Informationsfusion in der Mess- und Sensortechnik*, pp. 145–157.

Microsoft (2006, December). Microsoft robotics studio now available to provide common development platform. http://www.microsoft.com/presspass/press/2006/dec06/12-12msroboticsstudioavailablepr.mspx. Press Release.

Miene, A., U. Visser, and O. Herzog (2003). Recognition and prediction of motion situations based on a qualitative motion description. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida (Eds.), *RoboCup 2003: Robot Soccer World Cup VII*, Volume 3020 of *Lecture Notes in Computer Science*. Springer Verlag.

Miller, R. and M. Shanahan (1999). The event calculus in classical logic - alternative axiomatisations. *Electronic Transactions on Artificial Intelligence 3*, 77–105.

Milstein, A., JavierSànchez, and E. Williamson (2002). Robust global localization using clustered particle filtering. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02) and Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-02)*, pp. 581–586. AAAI Press.

Monahan, G. (1982). A survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science 28*(1), 1–16.

Montemerlo, M., S. Thrun, H. Dahlkamp, D. Stavens, and S. Strohband (2006). Winning the DARPA grand challenge with an AI robot. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)and the Eighteenth Innovative Applications of Artificial Intelligence Conference (IAAI-06)*. AAAI Press.

Moore, R. C. (1985). A formal theory of knowledge and action. In J. Hobbs and R. Moore (Eds.), *Formal Theories of the Commonsense World*, Chapter 9, pp. 319–358. Ablex Publishing Corp.

Moravec, H. and A. Elfes (1985). High resolution maps from wide angular sensors. In *Proceedings of the 1985 IEEE International Conference On Robotics and Automation (ICRA-85)*, pp. 116–121. IEEE Computer Society Press.

Mueller, E. (2006). *Commonsense Reasoning*. Morgan Kaufmann.

Munoz-Avila, H. and T. Fisher (2004). Strategic planning for unreal tournament bots. In *AAAI Workshop on Challenges in Game AI*. AAAI-04.

Murphy, R. (2000). *Introduction to AI Robotics*. The MIT Press.

Murray, J., O. Obst, and F. Stolzenburg (2001). Towards a logical approach for soccer agents engineering. In P. Stone, T. R. Balch, and G. K. Kraetzschmar (Eds.), *RoboCup 2000: Robot Soccer World Cup IV*, Volume 2019 of *Lecture Notes in Computer Science*, pp. 199–208. Springer.

Musliner, D., J. Hendler, A. Agrawala, E. Durfee, J. Strosnider, and C. Paul (1995). The challenges of real-time AI. *Computer 28*(1), 58–66.

Myers, K. (1996). A procedural knowledge approach to task-level control. In B. Drabble (Ed.), *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 158–165. AAAI Press.

Nardi, D., M. Riedmiller, C. Sammut, and J. Santos-Victor (Eds.) (2005). *RoboCup 2004: Robot Soccer World Cup VIII*, Volume 3276 of *Lecture Notes in Computer Science*. Springer.

Nau, D., T.-C. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman (2003). SHOP2: An HTN planning system. *Journal of Artifcial Intelligence Research (JAIR) 20*, 379–404.

Nejati, N., P. Langley, and T. Könik (2006). Learning hierarchical task networks by observation. In W. Cohen and A. Moore (Eds.), *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML-06)*, pp. 665–672. ACM.

Nilson, N. (1984). Shakey the robot. Technical Report 323, AI Center, SRI International.

Noda, I., H. Matsubara, K. Hiraki, and I. Frank (1997). Soccer Server: A Tool for Research on Multi-Agent Systems. *Applied Artificial Intelligence 12*(2), 233–250.

Obst, O. and J. Boedecker (2006). Flexible coordination of multiagent team behavior using HTN planning. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi (Eds.), *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*, pp. 521–528. Springer.

Obst, O. and M. Rollmann (2004, September). SPARK – A Generic Simulator for Physical Multiagent Simulations. In G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland (Eds.), *Multiagent System Technologies – Proceedings of the MATES 2004*, Volume 3187 of *Lecture Notes of Artificial Intelligence*, pp. 243–257. Springer.

Ögren, P. and N. Leonard (2005). A convergent dynamic window approach to obstacle avoidance. *IEEE Transactions on Robotics and Automation 21*(2), 188–195.

Orocos (2007). The ocoros project – smarter control in robotics & automation! http://www.orocos.org/. last visited Jan. 2007.

Parr, R. and S. Russell (1995). Approximating optimal policies for partially observable stochastic domains. In C. Mellish (Ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*. Morgan Kaufmann.

Parr, R. and S. Russell (1997). Reinforcement learning with hierarchies of machines. In M. I. Jordan, M. J. Kearns, and S. A. Solla (Eds.), *Advances in Neural Information Processing Systems*, Volume 10. The MIT Press.

Pednault, E. P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. In R. Brachman, H. Levesque, and R. Reiter (Eds.), *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pp. 324–332. Morgan Kaufmann.

Peitersen, B. and J. Bangsbo (2000). *Soccer Systems & Strategies*. Human Kinetics.

Penberthy, J. S. and D. S. Weld (1992). UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout (Eds.), *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pp. 103–114. Morgan Kaufmann.

Pham, H. (2006). Applying DTGolog to large-scale domains. Master's thesis, Department of Electrical and Computer Engineering, Ryerson University, Toronot, Canada.

Pineau, J., M. Montemerlo, M. E. Pollack, N. Roy, and S. Thrun (2003). Towards robotic assistants in nursing homes: Challenges and results. *Robotics and Autonomous Systems 42*(3-4), 271–281.

Pinheiro, P. and P. Lima (2004). Bayesian sensor fusion for cooperative object localization and world modeling. In *Proceedings of the Eighth Conference on Intelligent Autonomous Systems (IAS-04)*. Springer.

Pinto, J. (1998). Integrating discrete and continuous change in a logical framework. *Computational Intelligence 14*, 39–88.

Pinto, J. and R. Reiter (1993). Temporal reasoning in logic programming: A case for the situation calculus. In D. Warren (Ed.), *Proceedings of the Tenth International Conference on Logic Programming (ICLP-93)*, pp. 203–221. The MIT Press.

Pinto, J. and R. Reiter (1995). Reasoning about time in the situation calculus. *Annals of Mathematics and Artificial Intelligence 14*(2-4), 251–268.

Pinto, J., A. Sernadas, C. Sernadas, and P. Mateus (2000). Non-determinism and uncertainty in the situation calculus. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 8*(2), 127–150.

Pirri, F. and R. Reiter (1999). Some contributions to the metatheory of the situation calculus. *Journal of the ACM 46*(3), 325–361.

Polani, D., B. Browning, A. Bonarini, and K. Yoshida (Eds.) (2004). *RoboCup 2003: Robot Soccer World Cup VII*, Volume 3020 of *Lecture Notes in Computer Science*. Springer.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94*(1-2), 7–56.

Pradalier, C., J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier (2005). The cycab: a car-like robot navigating autonomously and safely among pedestrians. *Robotics and Autonomous Systems 50*(1), 51–68.

Precup, D., R. Sutton, and S. Singh (1998). Theoretical results on reinforcement learning with temporally abstract options. In C. Nedellec and C. Rouveirol (Eds.), *Proceedings of the 10th European Conference on Machine Learning (EMCL-98)*, Volume 1398 of *Lecture Notes in Computer Science*, pp. 382–393. Springer.

Preparata, F. P. and M. I. Shamos (1985). *Computational geometry: An Introduction*. Springer.

Puterman, M. (1994). *Markov Decision Processes: Discrete Dynamic Programming*. New York: Wiley.

Puterman, M. and M. Shin (1978). Modified policy iteration algorithms for discounted markov decision problems. *Management Science 24*(11), 1127–1137.

Quinlan, J. (1993). *C4.5 Programs for Machine Learning*. Morgan Kaufmann.

Qureshi, F., D. Terzopoulos, and R. Gillett (2004). The cognitive controller: a hybrid, deliberative/reactive control architecture for autonomous robots. In *IEA/AIE'2004: Proceedings of the Seventeenth International Conference on Innovations in Applied Artificial Intelligence*, pp. 1102–1111. Springer Springer Verlag Inc.

Rabin, S. (Ed.) (2002 & 2003). *AI Wisdom*, Volume 1 & 2. B & T.

Randell, D. A., Z. Cui, and A. Cohn (1992). A Spatial Logic Based on Regions and Connection. In B. Nebel, C. Rich, and W. Swartout (Eds.), *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pp. 165–176. Morgan Kaufmann.

Reiter, R. (1991). The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz (Ed.), *Artificial Intelligence and Mathematic Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press.

Reiter, R. (2001). *Knowledge in Action*. MIT Press.

Renz, J. and B. Nebel (1999). On the complexity of qualitative spatial reasoning: a maximal tractable fragment of the region connection calculus. *Artificial Intelligence 108*(1-2), 69–123.

Riedmiller, M. and A. Merke (2002). Using machine learning techniques in complex multi-agent domains. In I. Stamatescu, W. Menzel, and U. Ratsch (Eds.), *Perspectives on Adaptivity and Learning*. Springer.

RoboCup (2006). The Robocup Federation. http://www.robocup.org.

Röfer, T., I. Dahm, U. Dffert, J. Hoffmann, M. Jngel, M. Kallnik, M. Lötzsch, M. Risler, M. Stelzer, and J. Ziegler (2004). Germanteam 2003. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida (Eds.), *RoboCup 2003: Robot Soccer World Cup VII*, Volume 3020 of *Lecture Notes in Computer Science*. Springer.

Rosenschein, J. and M. Genesereth (1988). Deals among rational agents. In A. Bond and L. Gasser (Eds.), *Readings in Distributed Artificial Intelligence*, pp. 227–234. Morgan Kaufmann Publishers Inc.

Roth, P. M., H. Grabner, D. Skocaj, H. Bischof, and A. Leonardis (2005). Conservative visual learning for object detection with minimal hand labeling effort. In W. Kropatsch, R. Sablatnig, and A. Hanbury (Eds.), *Proceedings of the Twenty-Seventh DAGM Symposium for Pattern Recognition (DAGM-05)*, Volume 3663 of *Lecture Notes in Computer Science*. Springer.

Russel, S. and P. Norvig (2003). *Artificial Intelligence – A Modern Approach* (2nd ed.). Prentice Hall.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence 5*(2), 115–135.

Sacerdoti, E. (1975). The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)*, pp. 206–214.

Sandewall, E. (1995). *Features and Fluents: The Representation of Knowledge about Dynamical Systems*. Oxford University Press.

Sandewall, E. (1998). Cognitive robotics logic and its metatheory: Features and fluents revisited. *Electronic Transactions on Artificial Intelligence 2*, 307–329.

Sardiña, S. (2001). Local conditional high-level robot programs. In *IJCAI Workshop on Nonmonotinic Reasoning, Action and Change (NRAC-01)*. IJCAI-01.

Scherl, R. and H. Levesque (1993). The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 689–697. AAAI Press / The MIT Press.

Scherl, R. and H. Levesque (2003). Knowledge, action, and the frame problem. *Artifical Intelligence. 144*(1-2), 1–39.

Schiffel, S. and M. Thielscher (2005). Interpreting golog programs in flux. In *The Seventh International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense-05)*.

Schiffel, S. and M. Thielscher (2006). Reconciling situation calculus and fluent calculus. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06) and the Eighteenth Innovative Applications of Artificial Intelligence Conference (IAAI-06)*. AAAI Press.

Schiffel, S. and M. Thielscher (2007). Automatic construction of a heuristic search function for general game playing. In *Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change (NRAC-07)*.

Schiffer, S. (2005). A qualitative worldmodel for autonomous soccer agents in the readylog framework. Diploma thesis, Knowledge-based Systems Group, Computer Science Department, RWTH Aachen University.

Schiffer, S., A. Ferrein, and G. Lakemeyer (2006a). Football is coming home. In X. Chen, W. Liu, and M.-A. Williams (Eds.), *Proceedings of the International Symposium on Practical Cognitive Agents and Robots (PCAR-06)*. University of Western Australia Press.

Schiffer, S., A. Ferrein, and G. Lakemeyer (2006b). Qualitative world models for soccer robots. In S. Wlfl and T. Mossakowski (Eds.), *Qualitative Constraint Calculi, Workshop at KI 2006, Bremen*, pp. 3–14.

Schiffer, S., T. Niemueller, and A. Ferrein (2006). AllemaniACs. http://robocup.rwth-aachen.de.

Shanahan, M. (1990). Representing continuous change in the event calculus. In *In Proceedings of the European Conference on Artificial Intelligence (ECAI-90)*, pp. 598–603.

Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Las of Inertia*. MIT Press.

Shanahan, M. (2000). An abductive event calculus planner. *Journal of Logic Programming 44*(1-3), 207–240.

Shanahan, M. and M. Witkowski (2001). High-level robot control through logic. In C. Castelfranchi and Y. Lespérance (Eds.), *Intelligent Agents VII. Proceedings of the Seventh International Workshop Agent Theories Architectures and Languages, (ATAL-00) 2000,*, Volume 1986 of *Lecture Notes in Computer Science*, pp. 104–121. Springer.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence 60*(1), 51–92.

Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation 10*(1), 34–43.

Simmons, R. (1996). The curvature-velocity method for local obstacle avoidance. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation, (ICRA-96)*. IEEE Computer Society Press.

Simmons, R., R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, and M. Veloso (1997). Xavier: experience with a layered robot architecture. *SIGART Bulletin 8*(1-4), 22–33.

Simmons, R., R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan (1997). A layered architecture for office delivery robots. In L. Johnson (Ed.), *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*, pp. 245–252. ACM Press.

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

Son, T. and C. Baral (2001). Formalizing sensing actions: A transition function based approach. *Artificial Intelligence 125*(1-2), 19–91.

Sondik, E. (1971). *The Optimal Control of Parially Observable Markiv Processes*. Ph. D. thesis, Department of Electrical Engineering, Stanford University, Stanford, CA.

Sondik, E. (1978). The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operations Research 26*(2), 282–304.

Soutchanski, M. (2001). An on-line decision-theoretic Golog interpreter. In B. Nebel (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, (IJCAI-01)*. Morgan Kaufmann.

Soutchanski, M. (2003). *High-level Robot Programming in Dynamic and Incompletely known environments*. Ph. D. thesis, University of Toronto.

Soutchanski, M., H. Pham, and J. Mylopoulos (2006). Decision making in uncertain real-world domains using DT-golog. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso (Eds.), *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI-06), Including Prestigious Applications of Intelligent Systems (PAIS-06)*. IOS Press.

Stachniss, C. and W. Burgard (2002). An integrated approach to goal-directed obstace avoidance under dynamic constraints for dynamic environments. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-02)*. IEEE Computer Society Press.

Steinbauer, G., M. Faschinger, G. Fraser, A. Mühlenfeld, S. Richter, G. Wöber, and J. Wolf (2004). Mostly harmless team description. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida (Eds.), *RoboCup 2003: Robot Soccer World Cup VII*, Volume 3020 of *Lecture Notes in Computer Science*. Springer.

Steinbauer, G., J. Weber, and F. Wotawa (2005). From the real-world to its qualitative representation – practical lessons learned. In R. B., H. M., and W. F. (Eds.), *International Workshop on Qualitative Reasoning*, Graz, Austria, pp. 186–191.

Stolzenburg, F. and T. Arai (2003). From the specification of multiagent systems by statecharts to their formal analysis by model checking: Towards safety-critical applications. In M. Schillo, M. Klusch, J. P. Müller, and H. Tianfield (Eds.), *Proceedings of the First German Conference on Multiagent System Technologies (MATES-03)*, Volume 2831 of *Lecture Notes in Computer Science*, pp. 131–143. Springer.

Stolzenburg, F., O. Obst, and J. Murray (2002). Qualitative Velocity and Ball Interception. In M. Jarke, J. Koehler, and G. Lakemeyer (Eds.), *KI 2002: Advances in Artificial Intelligence, Proceedings of the Twenty-Fifth Annual German Conference on Artificial Intelligence, (KI-02)*, Volume 2479 of *Lecture Notes in Computer Science*, pp. 283–298. Springer.

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer (Intelligent Robotics and Autonomous Agents)*. MIT Press.

Strack, A. (2004). Robust self-localisation for mobile robots in the robocup domain. Diploma thesis, Knowledge-based Systems Group, Computer Science V, RWTH Aachen, Aachen, Germany.

Strack, A., A. Ferrein, and G. Lakemeyer (2006). Laser-based localization with sparse landmarks. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi (Eds.), *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020 of *Lecture Notes in Computer Science*, pp. 569–576. Springer.

Stroupe, A., M. Martin, and T. Balch (2001). Distributed sensor fusion for object position estimation by multi-robot systems. In IEEE (Ed.), *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA-01)*. IEEE Computer Society Press.

Stulp, F., S. Gedikli, and M. Beetz (2004). 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-04)*. KI-04.

Sutton, R. and A. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Sutton, R., D. Precup, and S. Singh (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence 112*(1-2), 181–211.

Tate, A. (1977). Generating project networks. In R. Reddy (Ed.), *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 888–893. William Kaufmann.

Thielscher, M. (1998). Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence 2*(3–4), 179–192.

Thielscher, M. (1999). From situation calculus to fluent calculus: state update axioms as a solution to the inferential frame problem. *Artificial Intelligence 111*(1-2), 277–299.

Thielscher, M. (2000). Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman (Eds.), *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR-00)*, pp. 109–120. Morgan Kaufmann.

Thielscher, M. (2001). The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence 131*(1-2), 1–37.

Thielscher, M. (2002a). Programming of reasoning and planning agents with FLUX. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams (Eds.), *Proceedings of Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*. Morgan Kaufmann.

Thielscher, M. (2002b). Reasoning about actions with CHRs and finite domain constraints. In P. Stuckey (Ed.), *Proceedings of Eighteenth International Conference on Logic Programming (ICLP-02)*, Volume 2401 of *Lecture Notes in Computer Science*. Springer.

Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming 5*(4-5), 533–565.

Thrun, S. (2006). Winning the DARPA grand challenge: A robot race through the mojave desert. In B. Werner (Ed.), *Proceedings of the Twenty-First IEEE/ACM International Conference on Automated Software Engineering (ASE-06)*, pp. 11. IEEE Computer Society Press.

Thrun, S., W. Burgard, and D. Fox (2005). *Probabilistic Robotics*. MIT Press.

Thrun, S., D. Fox, W. Burgard, and F. Dellaert (2000). Robust monte carlo localization for mobile robots. *Artificial Intelligence 128*(1-2), 99–141.

Thurau, C., C. Bauckhage, and G. Sagerer (2004). Synthesizing movements for computer game characters. In C. E. Rasmussen, H. H. Bülthoff, B. Schölkopf, and M. A. Giese (Eds.), *Proceedings of the Twenty-Sixth DAGM Syposium on Pattern Recognition (DAGM-04)*, Volume 3175 of *Lecture Notes in Computer Science*, pp. 179–186. Springer.

Tira-Thompson, E. (2004). Tekkotsu: A rapid development framework for robotics. Master's thesis, Robotics Institute, Carnegie Mellon University.

v. Waveren, J. (2001). The quake III arena bot. Master's thesis, University of Technology Delft, Faculty ITS, Delft.

Veloso, M., J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence 7*(1), 81–120.

von Neumann, J. and O. Morgenstern (1947). *The Theory of Games and Economic Behavior*. Princeton University Press.

Waldinger, R. (1977). Achieving several goals simultaneously. *Machine Intelligence 8*, 94–136.

Watkins, C. (1989). *Learning from Delayed Rewards*. Ph. D. thesis, King's College, Cambridge.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine 15*(4), 27–61.

White, J. (1999). Telescript technology: Mobile agents. In *Mobility: processes, computers, and agents*, pp. 460–493. ACM Press/Addison-Wesley Publishing.

Wilkins, D. (1990). Can AI planners solve practical problems? *Computational Intelligence 6*(4), 232–246.

WITAS (2007). http://www.ida.liu.se/ patdo/auttek/introduction/index.html. last visited January.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley & Sons.

Wooldridge, M. and N. R. Jennings (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*(2), 115–152.

Ziegelmeyer, D. (2006). Decision-theoretic planning in the dynamic logic ES. Diploma thesis, Knowledge-Based Systems Group, Computer Science Department, RWTH Aachen University.

Curriculum Vitae

Name	Alexander Antoine Ferrein
Geburtsdatum	9. Dezember 1974
Geburtsort	Hilden
Schulbildung	
1981 – 1984	Dreikönigen Schule, Neuss
1984 – 1991	Quirinus Gymnasium, Neuss
1991 – 1994	Pascal-Gymnasium, Grevenbroich
1994	Abitur
Studium	
1994 – 2001	Studium der Informatik an der RWTH Aachen
2001	Diplom
Tätigkeit	
2001 – 2008	wissenschaftlicher Mitarbeiter am Lehr- und Forschungsgebiet Informatik 5, Wissenbasierte Systeme RWTH Aachen