INTEGRATING DECISION-THEORETIC PLANNING AND PROGRAMMING FOR ROBOT CONTROL IN HIGHLY DYNAMIC DOMAINS

von
Christian Fritz
{Christian.Fritz@rwth-aachen.de}

Diplomarbeit

im Fach Informatik
vorgelegt an der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im Herbst 2003

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Acknowledgments

First of all I would like to thank our whole ROBOCUP team, the *AllemaniACs*, for the great work we did in preparation of, and the success we had at the ROBOCUP 2003 World Cup at Padua, Italy. I am especially grateful for being given the opportunity of applying the presented approach to our soccer playing robots at a world cup tournament, and thank everybody for trusting in this approach although it sometimes was rather unclear whether it would lead to success. Special thanks goes to Alexander Ferrein for all the ups and downs we went through and for never loosing faith in our success. Also for the endless and exhaustive discussions especially at the beginning of this work I would like to thank him. Last but not least, I want to thank Prof. Lakemeyer for the discussions concerning options and for giving me the opportunity to present parts of this work at an important international conference on artificial intelligence.

Contents

Chapter 1

Introduction

1.1 Autonomous Agents

The design of autonomous agents, like mobile robots, has become a key issue in artificial intelligence. Autonomous mobile robots are commonly able to fulfill certain so-called low-level tasks, such as collision avoidance or localization. Basic actions, as for example driving to a specified location, are thus possible for it to perform. In general, the designer's motivation is to use the robot to fulfill more complex tasks, e.g. to deliver mail or to play soccer. If the robot is technically capable of performing all necessary actions to carry out the task, the remaining question is *how* to tell the robot what it should do and how. The same question arises for non-physical robots, so that in our considerations we will talk more generally about an *agent*, denoting the concept of physical robots and non-physical agents that act in simulated worlds or any other kind of non-physical environment. An example of non-physical agents could be trading agents acting at on-line markets or stock exchanges or computer players in some kinds of games. The special term *autonomous agents* is used to point out that these agents are not controlled from the outside, but instead fulfill tasks on their own. Then a designer has to think of how to make these agents act *intelligently*.

Mainly, two paradigms for specifying the behavior of autonomous agents exist which are mostly treated independently: programming and planning.

The programming approach aims at specifying an agent through an explicit program. That is, the designer strictly defines the behavior of the agent. Imaginable would be a simple sequence of actions the agent performs or some use of conditionals to decide on different possible courses of action depending on some condition. Other control constructs include loops and procedures. After all, however, the behavior of the agent is completely decided by the designer. In particular, whatever situation the agent might get into, it just sticks to its program.

Here is the main limitation of this approach: as the complexity of the agent's task increases, the designing task for the agent programmer gets more and more difficult, meaning that there are more and more situations the programmer has to provide an appropriate action for. Still, during execution situations might be encountered that the programmer have not borne in mind. Then the agent would probably behave differently from what would be desired. Furthermore, even small changes in the agent's task may require a lot of modifications in the program. On the upside this approach is typically fast, as the agent usually does not have to "think" a lot.

Specifying an agent according to the planning paradigm is completely different. Instead of explicitly specifying how to behave, the agent is left with the decision on an appropriate course of action. In general, the designer defines only the actions and their effects, describes an initial state and a goal state. From that the agent is forced to create a *plan* how to reach the goal. This always involves some kind of search. Subtleties exist in various forms: systems exist that allow for uncertainty about the effects of actions, assigning certain probabilities to the outcomes considered possible. Others formulate goals not explicitly as some sort of desirable state to reach, but instead assign utilities to states such that the agent aims at maximizing the utilities along its path. Put together, those approaches constitute decision-theoretic planning, which will be the kind of planning considered in this work.

The planning approach overcomes the problem with programming mentioned above. As long as the models describing the effects of the actions are adequate[1] and the description of the goal and the initial state or the assignment of rewards to situations in the case of decision-theoretic planning correctly describe the desires of the programmer, the agent will always be able to behave as intended. However, this advantage comes along with a huge computational complexity. Instead of simply following the strict instructions given by the programmer, the agent has to *project* the effects of all possible courses of action up to a certain *horizon*. Only after these projections have been generated, the agent can determine the expected rewards for each of them and select the most promising for execution. Projection, however, takes time. The complexity depends directly on the number of possible actions determining the branching factor of the thereby defined *search tree* that is traversed in planning. Additionally, in domains with uncertainty, which mobile robotics certainly ranks among, the branching factor is further increased by the number of possible outcomes an action can have. This fact immediately restrains the application of planning in real-time systems when time to decide on the next action is limited.

Yet, there are approaches to combine the two paradigms. Boutilier et al. [5] recently proposed DTGOLOG. DTGOLOG combines programming and planning by integrating Markov Decision Processes [31] (MDPs) into the logic programming language GOLOG [25]. MDPs are commonly used in decision-theoretic planning for modeling the problem. GOLOG is based on the situation calculus [26], second order logic which can be used to model worlds where changes in the world are only due to actions. Besides these *primitive actions*, GOLOG offers common programming constructs such as conditionals, loops and procedures. Using the underlying situation calculus, projecting sequences of actions can be done very naturally which is the main benefit of GOLOG.

The idea by which DTGOLOG integrates planning into programming is that the user can leave certain decisions in his program up to the agent. At any point in the program the user may decide that instead of determining what to do, to enumerate a couple of alternatives and let the agent pick the *best* one. This can be very useful in cases when the designer has a good idea of the structure of the problem but not of an explicit solution. The structure of the problem is defined via models of the agent's actions plus some sort of reward to assess situations. The agent is then encouraged to find ways to reach situations with high reward. Seen from the point of view of programming, DTGOLOG introduces non-determinism with an implicit optimization semantics. From the perspective of planning, on the other hand, it enables the designer

[1] Problems in designing models arise for example from the qualification- and the frame problem which we will discuss in Section 3 and Section 4.1, respectively.

to restrain the set of possible plans and thereby decrease the time needed to find out which is the best.

However, DTGOLOG has some shortcomings which severely limit its use for realistic applications. The only implemented example reported in [5] was very simple in the way that the use of non-determinism was kept to a minimum. In larger planning problems the performance of DTGOLOG in solving the modeled MDP is poor compared to state-of-the-art algorithms of MDP literature, which themselves are computationally expensive. Another disadvantage is that DTGOLOG is an off-line interpreter. Programs have to be interpreted to the end before any action can be performed in the real world. Also sensing actions, actions querying sensor hardware and reporting the results, are not supported.

Fortunately, some of these disadvantages have already been discussed in the literature. One idea of speeding up planning in MDPs is to introduce so called *options* as discusses for example in [38, 21]. The idea is to increase the planning granularity by performing the planning not over primitive actions but over more complex actions. These complex actions, denoted options, are created using primitive actions and having the models of these primitive actions, models about the new options can be generated. Reducing the planning granularity by planning over options decreases the necessary horizon and may, depending on the problem and the implemented option, also help reduce the branching factor of the search tree. This way even exponential speed-ups are possible.

On the other hand, various on-line extensions of GOLOG have been developed and successfully tested [12, 18, 22]. These base on an incremental interpretation of programs, performing basic actions immediately when interpreted. Also sensing is supported. In particular, we will base our considerations and also the implementations on ICPGOLOG [22], which also includes some other useful extensions of GOLOG.

1.2 Goals and Contribution

The goal of this work it to tackle the disadvantages of DTGOLOG so that it can be applied to realistic and highly dynamic domains. Therefore, a new language is developed and an interpreter is implemented in ECLiPSe Prolog [10] and evaluated in three example domains.

1.2.1 Extending DTGolog with Options

As a first step, the idea of options is introduced into DTGOLOG [16]. The user is able to define so-called local MDPs to describe sub-problems. These MDPs are in turn solved producing a policy for the sub-problem. Such policies are called options. Further, for each such option a model is created which describes the effects of following this policy depending on the current situation. This enables the user to use options just like primitive actions. In particular, it is possible to, in turn, use options in describing other sub-problems via another local MDP. Thus, it is possible to create hierarchies of options, abstracting from the original fine-grained problem further and further.

In the MDP literature there are different suggestions how options can alleviate the global task. All these have in common that a new MDP is defined and differ only on the therein used set of states and actions. But by fixing the MDP in which the options are used, a certain amount of flexibility is lost. Yet, in our system, due to the flexibility in combining programming and planning, the user can decide freely how to use options.

This way, knowledge about the structure of the problem can be applied to further prune the search tree.

1.2.2 Merging extended DTGolog and icpGolog to ReadyLog

As we pointed out, we are interested in an on-line interpreter. One way of accomplishing this, is to extend DTGOLOG to an on-line interpreter like Soutchanski [37]. However, we instead aim at extending an existing on-line interpreter with the key features of DTGOLOG. We choose ICPGOLOG [22] for this. The reason is simple: ICPGOLOG already comprises many useful features that have been developed to enhance GOLOG. These are mainly the following: concurrency [13], continuous change [19], probabilistic projection of plans [20], and the progression of the knowledge base [15, 22].

De Giacomo and Levesque [12] discuss the problems of large agent programs containing both nondeterminism and sensing. They argue why an on-line execution style seems the only reasonable way of dealing with such programs. In this execution style actions get executed in the real world before advancing in the interpretation of the program. Then it is possible to react to the results of sensing actions. To still allow nondeterminism, which requires some kind of projection to make a reasonable decision between the alternatives, they introduced an off-line search operator Σ which can do just that. If applied to a program the operator searches for an execution trace to a successful termination of it. Although we are not simply interested in finding any way of terminating the program but finding the best one, the context stays the same: potentially large programs containing both nondeterminism and sensing. Thus, we want to go the same way of supporting nondeterminism in our programs, though, with a different semantics of finding the best choice. Hence, we will introduce new operators similar to Σ.

As a side condition, we want our interpreter to be fast enough to even in highly dynamic domains allow the use of nondeterminism in more than just a trivial amount.

1.2.3 Evaluation of the interpreter

In the following example domains the interpreter is evaluated. The selection of these example domains aims at setting up a broad range of different conditions and requirements set at the interpreter.

1. *Grid worlds*
   Commonly considered in MDP literature are the so called grid worlds. These are *discrete* navigation problems, where an agent living in a grid of cells is principally able to move to adjacent grid cells. Walls between cells or obstacles occupying them may exist to prevent moves between these cells. A common task for the agent would then be to find a shortest way from a certain initial cell to some kind of goal cell. Sometimes properties are assigned to some of the cells to make them behave differently from normal or additional rewards and costs are assigned to cells and actions.

   Grid worlds have the advantage of being simple enough so that new ideas can easily be tested. For instance, the idea of options was, to the best of our knowledge, yet only applied to grid world examples.

2. *RoboCup Soccer Simulation*
   The ROBOCUP soccer server [27] is a simulation software to simulate two teams of eleven software agents playing a soccer match against each other. A more detailed introduction to the simulator will be given in Chapter 2. The reason for us to choose this domain to be in our set of examples is because of its extremely high dynamics, it is a continuous world and most sensor information and actuator effects are uncertain. The task an agent faces in this domain is further complicated by the adversarial and cooperative component of having opponents and teammates.

   We want to use this domain to test the applicability in highly dynamic environments, where decisions have to be taken quickly and the state of the world changes rapidly and often in an unexpected way. *Real-time* decision making is needed.

3. *Mobile robotics*
   Here especially the ROBOCUP Mid-Size League will be considered. In this league physical robots play in teams of four and the field is approximately five times ten meters. More details about this league will be given in Section 2.

   This is an example of the kind of domains we are interested in after all: a highly dynamic domain where physical agents act autonomously.

1.3 Outline of this Thesis

The outline of this thesis is as follows. In the next chapter we will introduce our example domains in more detail. In Chapter 3 we will give an introduction to Markov Decision Processes and the concept of options in the literature. Chapter 4 describes the situation calculus and GOLOG. The most relevant GOLOG extensions are also introduced. Chapter 5 introduces READYLOG and its components in detail. Also, some implementational issues of the interpreter are discussed. Chapter 6 presents experimental results with the new interpreter in the example domains. We will show how options can be used to save exponentially in time, and how the new interpreter can even in very complex domains like ROBOCUP Simulation and mobile robotics be used to control agents successfully. We conclude in Chapter 7 and point out possible future work.

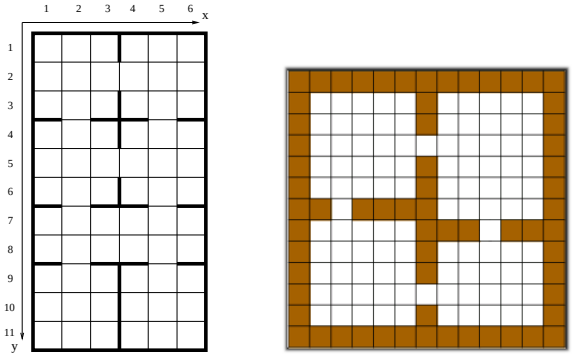
Chapter 2

Example Domains

In this chapter we introduce our example domains. They are used to evaluate the interpreter under different conditions and were choosen as to maximize the scope of possible requirements at the interpreter. Throughout this thesis we will show examples in these domains for illustration.

2.1 Grid Worlds

Grid worlds are virtual environments defined by a grid of cells representing locations. In these domains agents are able to move to adjacent cells as long as these are not separated by a wall and are not occupied. Figure 2.1 shows two example grid world. They are composed by different “rooms” which are connected by “doors”.



(a) (b)

Figure 2.1: (a) The Maze66 example taken from Hauskrecht et al. [21] together with the coordinate system we use. (b) The maze frequently used by Sutton et al. [29, 30, 38]

These kinds of worlds are commonly used for examples in MDP literature. In general, the task an agent faces in such a environment is a navigation problem, i.e. finding the best way from one specified initial cell to another specified cell, often denoted as

the goal cell.[1] The main characteristic of this domain is that it is discrete. The environment itself usually does not change, thus different situations are distinguished by the position of the agent in the grid.

2.2 ROBOCUP Soccer Simulation

The ROBOCUP is an international initiative to foster education and research in the areas of artificial intelligence and mobile robotics. By providing a standardized environment it is possible to compare different approaches to the problems arising. The problem given in the ROBOCUP domain is that of playing soccer in a team of agents competing against another team.[2] There are different leagues for different kinds of agents: For example, in the Sony Legged Robot League the teams consist of four Sony Aibo dog robots playing on a field of approximately two times four meters. For our research we consider the Simulation and the Mid-Size League, which we will describe in more detail in the following. In all leagues the robots are completely autonomous, i.e. there is in particular no human interaction during the games except for cases of malfunctioning of robots. Annually there is a world cup and several regional tournaments where the research groups meet to test and evaluate their approaches against each other.

In the simulation league the teams consist of eleven software agents playing in a simulated environment, called the soccer server [27]. A match lasts two times five minutes. The rules are mainly the same as in human soccer, taken from the official FIFA rules, with off-sides, throw-ins, corner kicks and so on. Additional rules regulate all other aspects of the simulation not present in human soccer. In particular, although several software agents may run on the same machine, it is forbidden to use inter-process communication. Via the soccer server the agents can broadcast (shout) short messages which can be heard by surrounding players. But this way of communicating is very limited: In particular, it is not possible to let one agent create an entire team strategy and command all other players accordingly. As a consequence each player has to make his own decision without usually knowing about the intentions of his teammates.

2.2.1 Soccer Server

The soccer server simulates a pitch of 105 times 68 meters. The players positions are represented as triples (x, y, θ) with x and y being the positions on the field with a precision of 0.1 meters, i.e. the field is discretized at steps of 10cm, and θ the agents angle with precision 0.1 degree. Similarly the ball position comprises x and y. Assigned with each object is further a velocity vector denoting the speed in x and y direction. The server does not broadcast any position informations to the agent programs. Instead, only the visual information each agent is able to perceive (see) from its current position is provided. Thus, the agent knows of the relative positions of other players, goalposts and some additional markers which are placed around the field. Using this, the agent can approximate its own position. The agent can only see objects within a certain cone in front and up to a certain distance. To look into another direction he can turn his entire body and/or his neck up to 90 degrees left or right from his current body

[1] In Chapter 3, when introducing MDPs more in detail, we will see that the best way not necessarily has to be the shortest way.

[2] In fact, the ROBOCUP initiative now also includes the so called ROBOCUP RESCUE where the task is to conduct rescue missions in some kind of catastrophe scenario. Since so far we are only concerned with the soccer leagues we will not further describe the other leagues.

angle. Figure 2.2 shows an example screen shot of the soccer monitor, which is used to constantly view the current situation of the soccer server, i.e. watch the game.

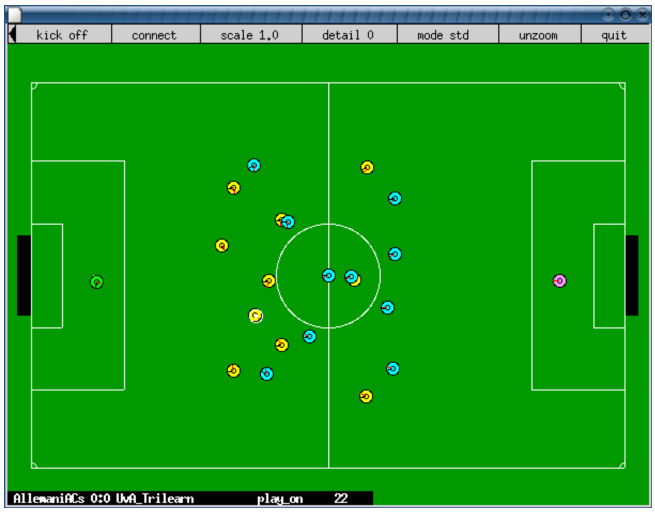


Figure 2.2: Screen shot of the soccer monitor. In the example our team (*AllemaniACs*) plays again an older version of the 2003 world champion *UvA-Trilearn*

Ignoring the neck angles and velocities there are $(680 \cdot 1050 \cdot 3600)^{22} \cdot 680 \cdot 1050 \approx 7.5 \cdot 10^{212}$ different situations. This is the number of possible (x, y, θ)-positions a player can be at $(680 \cdot 1050 \cdot 3600)$, to the power of the number of players (22), times the number of possible positions for the ball $(680 \cdot 1050)$. Since this is way to big to take as a state space for methods requiring a finite search space, like many planning algorithms do. But, because of the high precision of the positions, the domain can be treated as continuous.

During play an agent can perform the following base actions[3], which amounts to sending the soccer server a message stating his wish to perform such an action:

- `(dash Power)` – accelerate in forward ($Power > 0$) or backward ($Power < 0$) direction with a certain power
- `(turn Moment)` – turn by a certain moment
- `(kick Power Direction)` – kick the ball into a certain direction with a certain power (this action only affects the ball if it is in reach)
- `(turn_neck Angle)` – look into a direction relative to the body
- `(say Message)` – shout a message; note that the message size is currently restricted to 10 bytes

The effects of the first three actions are nondeterministic. In all cases noise is added to the parameters in form of a random number uniformly distributed over a certain range, which can be configured in the soccer server parameters files. The soccer server simulates the movement of objects according to their velocities at time steps (*cycles*) of

[3] we are leaving out some less important commands since they are irrelevant to our considerations

currently 100ms. Also this movement underlies uncertainty: for instance, in a situation at time t where the ball is at position (x_t, y_t) and moves with velocity (v_t^x, v_t^y), the position at the next time step t' will be $(x_{t'}, y_{t'}) = (x_t, y_t) + (v_t^x, v_t^y) + (\ r,\ r)$ with r being a random number whose distribution is uniform over a range around 0 depending on the current velocity. This noise constitutes the uncertainy of the system in the effects of actions and the future positions of moving objects.

Since immediately using the command set of the soccer server to control the players is tedious, it is common to create and use a base system which, based on the primitive server actions, offers more complex actions (often called skills, abilities or behaviors). Some of the research groups participating in the simulation league have published their base systems to help new teams in the league getting started ([9, 40]). For our work we have chosen to use the base system released 2002 by UvA Trilearn [40, 11], who won the World Cup at Padua (Italy) in 2003.

2.2.2 UvA Trilearn

The UvA Trilearn base system offers a world model, a set of hierarchical skills and implements the communication with the soccer server. In addition, if a player sees the ball, it immediately communicates its position to all teammates.

World Model

The world model comprises a large amount of data about the current and partially also on the last situation. In particular, the position and velocity of the agent himself and of all other players and the ball are provided. Usually the player will not see all other players and the ball. He then remembers the last known positions and velocities to estimate the positions of these objects. The so called confidence value for this object is then decreased to express the uncertainty whether the object still remains at this position/trajectory.

Apart from such world information, the world model also offers a number of functions to calculate different kinds of information based on the available data. For example, the closest opponent to the own player can be calculated or the expected position of the ball after a certain number of cycles in the future can be estimated.

Skills

The skills in the base system are divided into three levels of abstraction: low-level skills, intermediate level skills and high-level skills. The skills of each level are based on skills of any lower lever and the primitive soccer server actions. Also data from the world model is used.

The low-level skills work on primitive actions. Examples are

- `dashToPoint(pos)` – performs a `(dash Power)` such that the agent gets as close as possible to the position `pos`,
- `turnBodyToPoint(pos)` – performs a `(turn Moment)` such that the player afterwards faces position `pos`,
- `freezeBall` – performs a `(kick Power Direction)` such that the ball (if reachable) stops immediately. This is basically done by kicking in the inverse direction of the current ball movement.

Intermediate skills are more complex as they often involve different kinds of actions and last more than one cycle. The duration of skills however is not implemented explicitly, instead, such a skill has to be called over various cycles to reach its aim. The task of the skill, thus, is to recognize what to do next in order to reach its global aim and then to perform the corresponding action. This is repeatedly done based on the current world model only, i.e. ignoring all previous actions. For example the intermediate skill `moveToPos(posTo, angWhenToTurn)` performs the steps to take the player to a position `posTo`. To do so, in each cycle it first determines the angle between the body orientation and the direction to the target. Then, if the angle is absolutely less then `angWhenToTurn`, it performs a `dashToPoint(posTo)` or else calls `turnBodyToPoint(posTo)`.

Even more complex are the high-level skills. Here are some examples:

- `intercept` – intercept the (moving) ball,
- `dribble(angle)` – move with the ball into a certain direction,
- `directPass(pos)` – pass the ball towards a certain position.

It is possible to specify the agent behavior by only using the high-level skills together with information from the world model.

2.3 Mobile Robotics

In the domain of mobile robotics we consider mobile cognitive autonomous robots, that is robots that can freely change places in their environment (mobile), use sensors to perceive information from the world (cognitive) and react accordingly without immediate human interaction (autonomous). In this domain we again focus on ROBOCUP.

2.3.1 Mid-Size League

In the ROBOCUP Mid-Size league two teams of four robots measuring at most 40cm $\times$ 40cm $\times$ 80cm (width, length, height) play on a field of approximately five times ten meters. The current rules prescribe coloring one goal yellow and the other one blue. Poles at the corners of the field have similar color codings to make them easy to distinguish for vision systems. Some changes from the common sense soccer rules should be mentioned: throw-ins and corner-kicks do not exist in their common form, instead, when the ball moves out of bounds, the referee simply places the ball back onto the touch line. In particular, the team who forced the ball out of bounds may immediately take possession of the ball again. Charging, pushing opponent robots intentionally, is disallowed. This is especially reasonable as weight and power of the robots differ immensely. Stronger robots could otherwise simply push opponents away. Manual interference on the field and directly controlling robots remotely is strictly forbidden. In case of malfunctioning, robots may be taken out for repairs and be put back into play after at least 30 seconds have passed.

Different from the simulation league in the mid-size league the robots may communicate freely using wireless LAN or similar wireless communications. Also the set of allowed sensors is not strictly specified. Only satellite based localization (GPS) and changing the environment, for example by setting up active radio transmitters at certain points around the field, is disallowed.

Figure 2.3: Setting up for kick-off: our *AllemaniACs* mid-size team (in front) vs. *CoPS Stuttgart*

2.3.2 Architecture

Both hardware and software of the robots with which we are participating in the mid-size league have been developed at RWTH Aachen (University of Aachen).

Hardware

The hardware ([41]) has been developed by the Chair for Technical Computer Science ([36]). The aim was to have robots both competitive in ROBOCUP and usable for service robotics applications in office environments. Five robots were produced, one as substitute and one especially designed as a goal keeper. They are of size 39cm×39cm with height of approximately 55cm. Two modular Pentium III PCs at 933 MHz running Linux are on-board, the needed power is supplied by two lead-gel accumulators. These PCs are accessible by WLAN communication using the IEEE 802.11b standard at a maximal speed of 11Mbit/s. At the front side of the robots, some additional plates are mounted to improve controlling the ball while traveling with it.

The robots use the wheels and motors of an electronic wheel chair for moving (high-speed is 3m/s). A shooting mechanism at the front side can accelerate a ball which is not more then approximately 5cm away to a speed of about 2m/s. Also the following sensors are available:

- The *odometry* of the motors giving a good approximation of the distance each wheel moves,
- a 360 ° *laser range finder* which can run a scan resolution of 0.75 ° at a 20Hz frequency, providing the distances to any objects at height 28cm above the ground (= mounting height of the laser), and
- a *camera* on a pan-tilt unit.

Software

Our software consists of various modules. On one computer these communicate via a communication system using shared memory which we call *blackboard*. Via UDP communication, individual modules can remotely access a blackboard on a different computer. In this manner it is also possible to synchronize different blackboards.

The following modules are running on each robot during play:

Collision avoidance: takes coordinates relative to the robot, calculates the shortest collision-free trajectory to that target, and sends adequate commands to the motor to move along this path;

Localization: uses the distance measures of the laser scanner to estimate the current location of the robot on a given map of the environment;

Object tracking: from the distance measures and a map the robot finds objects that do not occur in the map and reports them as dynamic objects – this way we sense the position of opponent robots on the field;

Computer vision: from the images of the camera it tries to extract the ball and determine its relative position – this module runs alone on one of the two on-board computers connecting remotely to the other where all other modules are running;

Skill module: offers complex skills which form the set of actions being performed by the high-level controller like going to a certain position or performing a kick;

High-level control: this is where our interpreter is used to specify what the robot should do with regard to the actions available from the skill module.

On a control computer outside of the field certain data of each robot is collected and processed: Each robot reports its belief about its own position and whether and where it sees the ball. The different believes about the ball position are then fused into one global position estimate. This and the reported positions are then broadcast to all robots. Thus, even if a robot cannot see the ball for itself, it knows where it is expected as long as some teammate has spotted it. Furthermore, via the control computer we can communicate changes of so-called play modes (e.g. goals, game restarts, kick-offs..) to the robots which they are not able to notice themselves. This computer also allows us to watch and record the transmitted data. The hereby generated log-files can be replayed after the match to analyze the game itself and how the robots behaved in certain situations. This will be discussed in Chapter 6 in greater detail.

Chapter 3

Markov Decision Processes and Options

In realistic settings an agent almost never has complete knowledge about its environment. Thus, it has to act under uncertainty. This uncertainty can, for example, consist of not knowing which state the agent is in exactly or what the effects of performing a certain action will be. Under these circumstances it is not possible to guarantee that an agent will reach its goal with a certain plan. Sequential plans – a sequence of actions to take – are likely to fail. Instead conditional plans are more promising. This kind of plans include conditionals to react *on-line* to the actual state of the world: During plan execution, certain previously unknown details about the world are sensed and based on the sensing result a certain sub-plan is taken. For example consider the problem of going by car to a far city X, but you do not know how much gas you have left. Then a conditional plan like *"first check your gas, then, if it is enough, just drive to X, else, drive to a gas station, then refuel, then drive to X"* would seem appropriate. However, such a plan will not guarantee success in general, but only if certain assumptions are met, such as: the gas station has not run out of gas, my car has not been stolen, the road to city X has not been destroyed by an earthquake... This is called the *qualification problem* arising from the impossibility of stating all the preconditions under which an action will have its expected effect in the real world. However, such disqualifications should be assumed away as even trying to account for all of them would make the problem intractable. Imagine for the given task the agent would try to come up with a conditional plan like the following:

```
1   if (my car has been stolen)
2     then buy a new one, check the gas,
3          if (enough gas)
4            then if (road to X has been destroyed)
5                   then search new road to X, goto X,
6                   else goto X,
7            else if (gas station has run out of gas)
8                   then search gas station with gas,
9                        refuel, goto X,
10                  else refuel, goto X,
11  ...
```

Obviously, this would be complicating things more than necessary. However, the agent should be aware of this problem and be ready to re-plan in case of surprising events. We will discuss this issue in greater detail in Section 5.1.

3.1 Markov Decision Processes

Many of the problems of planning under uncertainty can be modeled as Markov Decision Processes (MDPs) [31] which also have become the standard model for decision-theoretic planning. We formally define an MDP as a tuple $M = \langle A, S, Tr, \mathcal{R} \rangle$, with:

A : a set of *actions*

S : a set of *states* (often called *state space*)

$Tr : S \times A \times S \rightarrow [0, 1]$ a *transition function*

$\mathcal{R} : S \times A \rightarrow \mathbb{R}$ a *reward function*

In the following we will describe these elements in more detail and explain certain assumptions we make about them together with their consequences.

3.1.1 Actions, States and Transitions

A state can be defined as a compact description of the world at a certain time-point. We assume that states comprise all relevant information about the world the agent needs for decision-making. The world is assumed to evolve in *stages*, which can be understood as time points. The transition function defines connections between the states of subsequent stages: for a state s_1 the transition function $Tr(s_1, a, s_2)$ defines the probability that the system changes to state s_2 after executing a given action a. Thus, for a stage t from the set of stages $\mathcal{T}$ it defines the probability $Pr(S^{t+1} = s_2 | S^t = s_1, A^t = a)$ where S^t is the state at stage t, S^{t+1} the state at stage $t + 1$ and A^t the action taken at stage t. This implies the *Markov assumption* that the next state only depends on the current state and the performed action. In particular, the history of states and actions are irrelevant for predicting the next state. Note, however, that information of earlier states may be included in the current state.

We take over the term *stage* used by [6] to denote the steps in which the system evolves. The transition of a stage t to a stage $t + 1$ is marked by an *event* such as an action taken by the agent (see Section 3.1.2). As we will not consider events that do not affect the state, we can equate stage transitions with state transitions. Assuming that no such event terminates instantaneously, stages can be thought of as different time points. We make the assumption that the stage does not influence state transitions. The model is then called *stationary*. Such an MDP can be depicted as a directed graph like in Figure 3.1.

Likewise, it is possible to represent the transition function Tr as a set of *transition matrices* Tr_{a_i}, one for each action $a_i \in A$. The entries of the matrix for action a_k would then be $p_{ij}^{a_k} = Tr(s_i, a_k, s_j)$.

We assume that not all actions are executable at every state, but still we do not distinguish over stages. Then, for each state $s \in S$ we get the *feasible set* $A_s \subset A$ of actions executable in that state. This is in analogy to action preconditions in other AI planning approaches. In our example MDP of Figure 3.1 there are four states $(s_1, .., s_4)$ and two different actions (a, b). In all states both actions are executable, except state

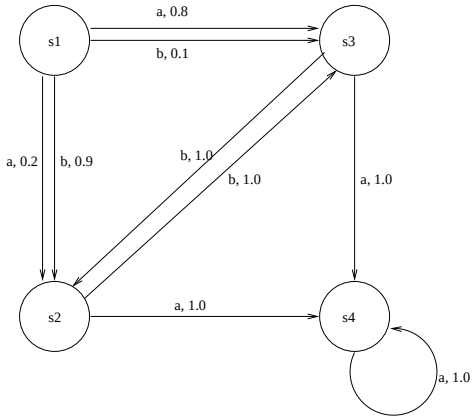


Figure 3.1: A simple example of a (stationary) MDP depicted as a graph. The arrows are labeled by actions and the probability for the transition to happen if performing this action.

s_4 which got $A_{s_4} = \{a\}$ as its feasible set. The row for s_4 in the transition matrix of action b then only contains 0.0's in contrast to rows of actions in the feasible set which always sum up to 1.0.

State s_4 is also special in the sense that it is an *absorbing state* saying that once you entered this state there is no way of leaving. More technically speaking $Tr(s_4, x, s) = 0.0 \forall x \in A_{s_4},\ s \neq s_4$. A subset $C \subset S$ which enjoys this property, that is, C has no transitions leading out of it with probability greater 0.0, is called *closed set*. It is called *proper closed set* or *recurrent class* if there is no proper subset of C which again is closed. Thus, in our example the set $C = \{s_2, s_3, s_4\}$ is a closed set, but not a proper closed set, since s_4 as an absorbing state forms a special case of closed set. States that do not belong to any proper closed set are called *transient*.

We are only interested in so called *fully observable* MDPs. In a fully observable MDP the agent always knows which state it is in. In particular, there is no uncertainty about his initial situation. In *partially observable* MDPs (POMDPs) the agent does not know the exact system state, but at each stage has a probability distribution over the state space. Although POMDPs are more general and even seem more appropriate for some of our domains, they are left out of consideration here as they would increase complexity and thus computational costs even more.

In the literature almost exclusively finite state and action sets are considered and the most popular and well investigated methods for solving MDPs (value iteration and policy iteration) require this property. Nevertheless we will also see an algorithm that, under certain circumstances, solves an MDP without this requirement (cf. Section 4.2.2).

3.1.2 Events

We think of an *event* as either an action taken by the agent or any *exogenous event* changing the system state and whose occurrence is not under the control of the agent. However, the probability of occurrence depending on the state might be known. An example could be an action taken by another agent or a natural process, such as a vase

which, after falling from a cupboard, breaks as it hits the floor. Although not controlled by the agent, events of this kind have to be taken into account for decision making.

In real world domains and, above all, multi-agent systems like ROBOCUP, exogenous events play an important role. Understanding how events can be modeled in the MDP context will help us in later sections to explain the reasons why uncertainty is modeled in a special form (cf. Section 5.1.1). We, therefore, discuss that topic relatively detailed here.

If a robot pushed a vase from the cupboard and this vase breaks on the floor, this could be seen as an effect of the robot's pushing action. Thus, the effect is composed of the immediate effect of pushing the vase from the cupboard and the subsequent events of falling and breaking. Such an action model is called an *implicit event model*, as the subsequent events are *implicitly* modeled in the robot's push action. A possible decomposition of this process could for example result in a robot push action, an event of falling taking place in the situation after pushing, and an event describing the eventual breaking at the moment the falling vase hits the ground. If the effects of an action are seen in this fashion, we speak of an *explicit event model*.

Although the explicit model seems to be more natural and is in general more intuitive to generate, for decision making the implicit model is needed. The reason is obvious: if a chosen action besides its intended effect also triggers another event or makes it more likely, the impact this event has on the success criteria has to be taken into account in the decision. Yet it is not generally easy to determine the transition for an action and a number of events, since their interaction can be rather complex. The order of temporal occurrence of action and events can influence the outcome: Imagine a mail delivery robot having a number of deterministic actions, one of which is checking the inbox for new mail. New mail may arrive at every stage with probability 0.1. Then the explicit event of mail arrival can be combined with the effects of the other actions to obtain an implicit event model. Thus, each action now has two possible outcomes: the original effect with probability 0.9 and the conjunction of the original effect together with new mail with probability 0.1. For any action not affecting the status of the inbox this is no problem. But consider the check-inbox action, which checks the inbox for new mail and if there is some takes it out for delivery. Here the way of combining the new mail event and the action is crucial for the outcome: if first the event happens and then the action, the robot would have mail and the inbox would be empty. On the other hand, if the action takes place before the event, the mail would remain in the inbox.

Even worse is the case where events can happen simultaneously. Then the outcome may not even be a sequence of the individual effects.[1] This problem is especially likely to occur in multi-agent systems with continuous time and where actions have a duration instead of terminating instantaneously.

One method to combine explicit events and actions is the following: For each event and action we specify transition probabilities for when they occur in isolation. This can be represented as a transition matrix as above. We do allow that an event does not change the state with some probability. If that is 1.0, we can think of it as the event not being possible in that state. Similarly, if this probability is, e.g. 0.7, we say that the event in this state only occurs with probability 0.3. In addition to these transition probabilities a *combination function* is required. As pointed out, this can get very complex or in the case of real simultaneous events even unrelated to the individual effects. Here we adopt an interleaving semantics for events and action (compare to [13]). Further we assume that events are *commutative*, that is, for every two events

[1] For an example see [6].

e_i, e_j and every state s applying e_i to s and then e_j has the same effect as first applying e_j and then e_i. The implicit transition probabilities can then simply be calculated by the multiplication of the independent transition matrices Tr_{e_i} and Tr'_a of the events and the action a, i.e. $Tr_a = Tr_{e_1} \cdot \ldots \cdot Tr_{e_n} \cdot Tr'_a$.

3.1.3 Costs, Reward and Value

The user's preferences are defined by the reward function $\mathcal{R} : S \times A \rightarrow \mathbb{R}$ which defines the desirability of executing a certain action in a certain state implying any possible reward for simply being at this state. To better understand this function and to alleviate its definition we separate it into two functions, a new reward function $R : S \rightarrow \mathbb{R}$ and a *cost function* $C : S \times A \rightarrow \mathbb{R}$. The new reward function is understood to express the desirability of being in a certain state, whereas C assigns (punitive) costs to actions depending on the state where they are executed.[2] Then the original reward function is simply $\mathcal{R}(s, a) = R(s) - C(s, a)$.[3]

For decision making the agent is interested in the overall quality of a sequence of actions, thereby passing by certain states. We define the *system history* at stage t as the sequence:

$$\langle\langle S^0, A^0\rangle, \ldots, \langle S^{t-1}, A^{t-1}\rangle, S_t\rangle^4$$

and denote the set of all system histories by $\mathcal{H}_S$. Then we can define a *value function* $V : \mathcal{H}_S \rightarrow \mathbb{R}$ to evaluate system histories. Commonly in MDP literature value functions are defined as the sum of all rewards and costs along the way as defined by the corresponding functions. Such a value function is called *time-separable* and *additive*, as it is a combination of the values accrued at each stage/time-point (time-separable) and this combination is a simple addition (additive). Then the value of a history h of length T is defined as ([4]):

$$V(h) = \sum_{t=0}^{T-1} [R(s^t) - C(s^t, a^t)] + R(s^T).$$

This is an evaluation for a course of action over T stages. T is called the *horizon* of the problem. We distinguish *finite-horizon problems*, where T is a natural number less then infinity, and *infinite-horizon problems*, with $T = \infty$. In infinite-horizon problems the value as defined above could be unbound, saying that a policy, if executed for long enough, can be infinitely good or bad. Since such a value function does not seem to be of much use and assuming that we prefer earlier rewards to later, it makes sense to introduce a *discount factor* $0 \leq \gamma < 1$ which is multiplied to rewards (and costs) of later stages. Then the value for such an *expected total discounted reward* problem is defined as ([4, 1]):

$$V(h) = \sum_{t=0}^{\infty} \gamma^t [R(s^t) - C(s^t, a^t)]$$

ensuring a bound value.

In addition to finite- and infinite-horizon problems there are so called *indefinite-horizon problems*. These are problems that terminate after a finite number of stages, but

[2] The function C is understood to assign only *action related* costs.

[3] Note that both R and C can be negative in which case their meaning is somewhat inverted (negative rewards are punitive, negative costs are beneficial).

[4] Since we are dealing with fully observable MDPs we can leave *observations* out of consideration

different from finite-horizon problems this number is not known in advance. Instead of simply breaking execution after a certain number of stages, in indefinite-horizon problems certain states are terminal (having an empty feasible set), preventing any further action and, thus, any reward gain or cost accumulation. It is required that one of these absorbing states is eventually reached with certainty from any state in the state space. This implies that there must not be any proper closed set (recurrent class) apart from the absorbing states. Indefinite-horizon problems are common in classical AI planning where only a certain goal is to be reached.

3.1.4 Policies and Expected Value

The decision problem the agent is facing is that of finding an optimal plan, that is one that maximizes the overall value. Such a plan at each stage can be conditioned on the system history until then. To capture the intention behind such a plan we define a *policy* π to be a mapping from the set of all system histories to actions, i.e. $\pi : \mathcal{H}_S \rightarrow A$. Intuitively, a policy π tells the agent for each possible system history what to do. Following a policy makes certain system histories more likely than others. Hence, it induces a probability distribution over system histories, $Pr(h|\pi)$. Then the *expected value of a policy* π is defined as:

$$\mathcal{E}(\pi) = \sum_{h \in \mathcal{H}_S} V(h) Pr(h|\pi).$$

The expected value of a policy can be used as a criterion to base the decision on: choose the policy that maximizes the expected value.

The set of system histories is *infinite* (as long as we do not set a limit on the length of the histories) which may lead to complex policies. Luckily, under the assumptions of full observability and a time-separable value function, the optimal action depends only on the current state and the stage. Consequently, policies can be represented in the much simpler form $\pi : S \times \mathcal{T} \rightarrow A$, that is, assigning each state-stage combination an action to execute in that case. Such a policy is also called *Markov policy*. Figure 3.2 illustrates how intuitive such a policy can be represented, when additionally stages are ignored.

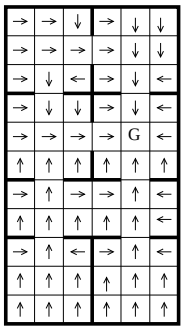


Figure 3.2: The arrows form a possible representation of a policy for the Maze66 where each action (right, left, down, up) has cost 1 and at G there is a high positive reward.

3.1.5 Solution Methods

Solving an MDP is understood to be the problem of finding a policy π that maximizes the expected value. Let a fully-observable MDP with a time-separable, additive value function be given, i.e. a state space S, an action space A, a transition function Tr, and a cost and a reward function C and R. Then, for a finite-horizon T and t the number of stages to go, we define the value function V_t^π for policy π as follows: set $V_0^\pi(s) = R(s)$ for all $s \in S$ and then:

$$V_t^\pi(s) = R(s) - C(s, \pi(s,t)) + \sum_{s' \in S} [Tr(s, \pi(s,t), s') V_{t-1}^\pi(s')] \tag{3.1}$$

Recall that π maps state-stage combinations to actions and thus $\pi(s,t)$ denotes an action. We can now define an optimal policy: A policy π is called *optimal* for horizon T if and only if $V_T^\pi(s) \geq V_T^{\pi'}(s)$ for all policies π' and all states $s \in S$. The value function of such an optimal policy π is called the *optimal value function*.

The most common algorithms for solving MDPs of that kind are value iteration and policy iteration. Both are dynamic programming approaches [4] exploiting the following property of the optimal value function:

$$V_t^*(s) = R(s) + \max_{a \in A} \{-C(s,a) + \sum_{s' \in S} [Tr(s,a,s') V_{t-1}^*(s')]\} \tag{3.2}$$

Value Iteration

Setting $V_0^*(s) = R(s), \forall s \in S$ the *value iteration algorithm* computes the optimal value functions for any $t > 0$ by iteratively applying Equation (3.2). From that, the elements $\pi(s,t)$ of an optimal policy π can be generated by taking any maximizing action a of the equation for value $V_t^*(s)$.

For infinite-horizon problems intuitively the stage does not matter for the decision as there are always infinitely many stages remaining. Indeed, Howard [1] showed that in such a case there is always an optimal *stationary* policy, i.e. a policy only depending on the state ($\pi : S \to A$). Then, with the discounted expected value as one's optimization criterion, the optimal value function satisfies the recurrence:

$$V^*(s) = R(s) + \max_{a \in A} \{-C(s,a) + \gamma \sum_{s' \in S} [Tr(s,a,s') V^*(s')]\}. \tag{3.3}$$

To generate an optimal policy in that case, one can use a slight modification of (3.2):

$$V_{t+1}(s) = R(s) + \max_{a \in A} \{-C(s,a) + \gamma \sum_{s' \in S} [Tr(s,a,s') V_t(s')].\} \tag{3.4}$$

For an arbitrary initial assignment V_0 the functions V_t converge for $t \to \infty$ linearly to the optimal value function V^* (see [31] for a proof).

For indefinite-horizon problems the same iteration procedure as for the infinite case can be applied.

Policy Iteration

Although we are not going to use policy iteration, we here describe it briefly for completeness. While value iteration aims at calculating an optimal value function and from

that extracts the optimal policy, the *policy iteration algorithm* directly operates on the policy. Beginning with an arbitrary policy[5] π_0 the iteration takes place in two steps:

1. *Policy evaluation*: compute the value function $V^{\pi_i}(s)$ for all $s \in S$
2. *Policy improvement*: at each state $s \in S$, find an action a^* as to maximize

$$Q_{i+1}(a,s) = R(s) - C(s,a) + \gamma \sum_{s' \in S} [Tr(s,a,s')V^{\pi_i}(s')]$$

and set $\pi_{i+1}(s) = a^*$.

The iteration eventually ends when $\forall s \in S\, .\pi_{i+1}(s) = \pi_i(s)$. The algorithm converges at least linearly to an optimal policy. For a further discussion of policy iteration and a comparison to value iteration, we refer to the literature ([31]).

3.2 Options

In artificial intelligence the need for hierarchical planning and abstraction from primitive actions has been recognized. In classical AI planning so called *macro operators* (or simply *macros*) have been investigated to enable reuse of sub-plans and raise the level of abstraction for planning. Macros, classically, are fixed sequences of actions that are considered for frequent use. That is, if sub-problems occur several times in related problems, a macro solving this sub-problem can be reused saving computational effort. Designing macros in a way that they can be used just as primitive actions, it is possible to hierarchically build macros over macros.

In stochastic settings, like the one's we are concerned with, simple sequences of actions (like macros) are not of much use as said at the beginning of this chapter. The term *option* (sometimes also *macro-action*) is used to denote a concept similar to a macro for stochastic environments and generalizes from action sequences to policies. In analogy to our discussion of sequential and conditional plans, options can be thought of as conditional sub-plans, whereas macros form sequential sub-plans. Hence, the advantages of conditional plans over sequential plans apply to options when compared to (sequential) macros.

To get across the intuition behind options we consider a first example for an option in the Maze66 environment.

Example 3.2.1 *Figure 3.3 shows the Maze66 environment with numbers assigned to the rooms. Assume we are generally interested in navigation problems in this domain. Then the agent will frequently be located in a room different from the room the goal is in and the agent's decision can be abstracted to deciding on the door by which to leave this room. After this decision being made, the agent only needs to find the best (e.g. shortest) way through the room to that door. Such a sub-plan forms an option. Thus, we could create two options for Room 1: one option taking the agent out to the right (Room 2), one option taking it to Room 3. Since options are entire policies instead of only sequential plans, they are applicable from every position in Room 1.*

In MDPs, options have been considered from different perspectives. Sutton et al. [29, 30, 2, 38] take a reinforcement learning point of view, whereas Hauskrecht et al. [21] continue Sutton's investigation focusing on planning with options.

[5] Note that policy iteration is applicable to infinite-horizon problems only. Hence, a policy can ignore the stage and, thus, has always size $|S|$.

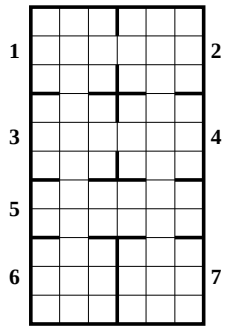


Figure 3.3: The Maze66 example with numbers assigned to the rooms.

3.2.1 Options and Multi-Time Models (Sutton et al.)

Sutton et al. [38] define an option O over an MDP $M = \langle A, S, Tr, \mathcal{R} \rangle$ as a tuple $\langle \mathcal{I}, \pi, \beta \rangle$, with:

- $\mathcal{I} \subseteq S$ the *initiation set*,
- $\pi : S \times A \rightarrow [0, 1]$ a policy, and
- $\beta : S \rightarrow [0, 1]$ a termination condition.

The initiation set determines the states where the option is applicable. The policy π defines for each state s a distribution $\pi(s, \cdot)$ over actions. On execution, the next action to take at a stage t is chosen according to $\pi(s_t, \cdot)$. The mapping β assigns to each state a probability that the option terminates if this state is reached. One natural assumption is that for all states s with $\beta(s) < 1.0$ this state is also included in $\mathcal{I}$. Consequently, any state s' outside of the initiation set ($s' \in S - \mathcal{I}$) would have a probability of termination equal 1.0 ($\beta(s') = 1.0$). Hence, it would suffice to define the policy π over $\mathcal{I}$ instead of over entire S.

The key insight of the work by Sutton and his colleagues is that with an appropriate transition model and a reward for an option, one can treat the option just like a primitive action. In particular, it can be used in planning. Sutton et al. call the models providing this information *multi-time models*. A multi-time model consists of a *reward prediction vector* $\mathbf{r}$ and a *state prediction matrix* $\mathbf{P}$. The vector $\mathbf{r}$ contains the truncated expected reward for each state $s \in S$, which is the discounted accumulated reward along the way when executing the option. Matrix $\mathbf{P}$ can be seen as a transition matrix, stating for all states the probabilities of ending in it when the option is executed in a certain state. Note that this prediction is not for one step, but for a yet unspecified duration: the execution of an option usually lasts several stages, where the exact number is not known in advance. Formally the elements of these predictions for an option o are defined as:

$$r_s^o = E\left\{ r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k} \mid \mathcal{E}(o, s, t) \right\}$$

$$p_{ss'}^o = \sum_{j=1}^{\infty} \gamma^j Pr\left\{ s_{t+k} = s', k = j | \mathcal{E}(o, s, t) \right\}$$

where $t + k$ denotes the random time of termination of the option, $\mathcal{E}(o, s, t)$ denotes the event of starting option o in state s at time t.

In [30] theoretical results of dynamic programming are extended for the use of options: Let $\mathcal{O}$ be a set of options – here options may also be primitive actions. $\mathcal{O}_s$ denotes the set of options available in state s. The set of possible policies over options, which are mappings defining the probabilities of taking a certain option in a certain state, is denoted $\Pi_{\mathcal{O}}$. Then the authors define the *optimal value function, given the set* $\mathcal{O}$, as:

$$V^*_{\mathcal{O}}(s) = \sup_{\pi \in \Pi_{\mathcal{O}}} V^{\pi}(s), \qquad \forall s \in S.$$

The authors show the following: The value function for any Markov policy $\pi \in \Pi_{\mathcal{O}}$ satisfies the Bellman evaluation equations:

$$V^{\pi}(s) = \sum_{o \in \mathcal{O}_s} \pi(s,o) \left(\mathbf{r}_o(s) + \mathbf{P}_o(s) \cdot \mathbf{V}^{\pi} \right), \qquad \forall s \in S,$$

where the policy $\pi(s,o)$ states the probability with which option o is chosen in state s, $\mathbf{r}_o(s)$ is the entry for state s in the reward prediction vector and $\mathbf{P}_o(s)$ similarly the corresponding row in the state prediction matrix for option o. $\mathbf{V}$ is the vector notation for the value function V, defined by $\mathbf{V}[i] = V(s_i)$, where $\mathbf{V}[i]$ denotes the i-th entry of the vector. The thereby defined system of equations then has the vector $\mathbf{V}^{\pi}$ as its unique solution. Further, the value function also satisfies the Bellman optimality equations:

$$V^*_{\mathcal{O}}(s) = \max_{o \in \mathcal{O}_s} \left\{ \mathbf{r}_o(s) + \mathbf{P}_o(s) \cdot \mathbf{V}^*_{\mathcal{O}} \right\}, \qquad \forall s \in S.$$

Here again $V^*_{\mathcal{O}}(s)$ is the unique solution. Also it is shown that there exists at least one optimal policy π^*, defined as a policy whose value function is optimal, i.e. $V^{\pi^*}(s) = V^*_{\mathcal{O}}(s)$.

An essential role in the proof of these results plays a theorem about the relationship between the model of a composed option and the models of its component:

Theorem 3.2.1 (Composition Theorem) *Given two options a and b together with their models $\mathbf{r}_a$, $\mathbf{P}_a$ and $\mathbf{r}_b$, $\mathbf{P}_b$, then for all states s:*

$$\mathbf{r}_{ab}(s) = \mathbf{r}_a + \mathbf{P}_a(s) \cdot \mathbf{r}_b$$

$$\mathbf{P}_{ab}(s) = \mathbf{P}_a(s) \mathbf{P}_b$$

where ab denotes the composed option of first performing option a and then option b.

From these results it follows that known algorithms like value iteration are applicable for computing value functions also for a given set of options. This is the key to planning with models of options and forms the theoretical basis for all our further considerations of options.

The examples presented in [30] consider the use of options *together* with primitive actions. They show how this speeds up convergence of value iteration. Figure 3.4 compares value iteration using only primitive actions (top) and using options (including primitive actions) (bottom). Here, similar to our previous example, options have been defined for leaving each room through a certain door. Such an options is defined as $\mathcal{I} =$ "all states in room X", π an appropriate policy (e.g. like in Figure 3.2), β("hallway states") $= 1.0$ and $\beta(s) = 0.0$ for all other states s. As options in one step can provide values for an entire rooms, already after the second iteration there have been values assigned to all states. Usual value iteration over primitive actions, on the other hand, has until then only reached states at distance two from the goal.

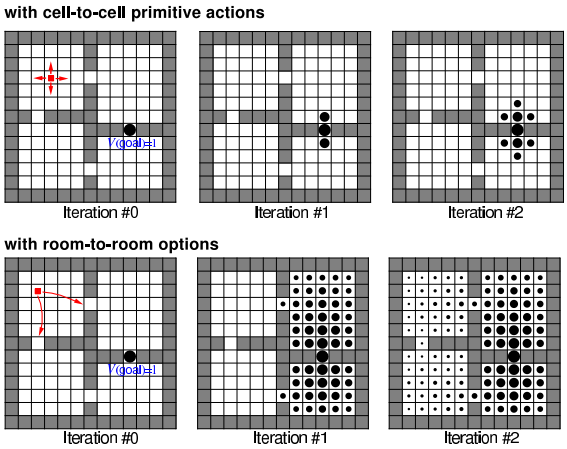


Figure 3.4: Taken from [2]: comparing value iteration with and without options. The size of the dots represent the value of the corresponding state (if yet any is assigned).

3.2.2 Abstract MDPs (Hauskrecht et al.)

Inspired by the work of Sutton and his colleagues, Hauskrecht et al. [21] further investigated the use of options (which they call macro-actions) for planning. For solving large MDPs they propose an hierarchical model using an *abstract MDP* which abstracts from the original state space reducing its size significantly. Since our own work is closely related to the investigations of this group we will present their approach more in detail.

Hauskrecht et al. define a *macro-action* simply as a policy for a certain region (subset of the state space). This policy can then, intuitively, be executed within this region and terminates as soon as leaving it. With regard to the more general definition of options by Sutton et al., a macro-action can be defined as an option $\mathcal{O} = \langle \mathcal{I}, \pi, \beta \rangle$, where:

- $\beta(s) = \begin{cases} 0.0, & s \in \mathcal{I} \\ 1.0, & otherwise \end{cases}$
- $\pi : \mathcal{I} \times A \rightarrow \{0, 1\}$ and can thus be represented as $\pi : \mathcal{I} \rightarrow A$.

The approach of [21] relies on a *region-based decomposition* of the MDP. This is basically a partitioning $\Pi = \{S_1, \ldots, S_n\}$ of state space S, where the S_i are called the *regions* of the MDP. Furthermore, *exit states* and *entrance states* for a region are defined. The set of these states are called the *exit periphery* ($XPer(S_i)$) and the *entrance periphery* ($EPer(S_i)$), respectively. Intuitively, the exit periphery of a region is the set of those states outside the region which can be reached from inside by some action with probability greater zero. Similarly, the entrance periphery consists of all states inside a region reachable from the outside. Figure 3.5 shows the set of peripheral states marked by gray dots for a decomposition of the Maze66 example environment where each room defines a region.

Based on that, the required models for planning can be defined (taken from [21] to our notation):

Definition 3.2.1 *A* discounted transition model $Tr_i(\cdot, \pi_i, \cdot)$ *for a macro-action* π_i

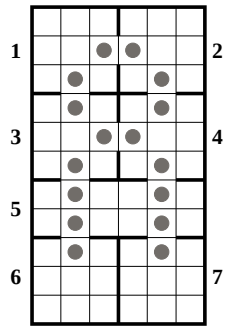


Figure 3.5: The periphery (set of all peripheral states) marked by gray dots for a room-oriented decomposition of the Maze66 environment – each room equals one region.

(defined on region S_i) is a mapping $Tr_i : S_i \times XPer(S_i) \rightarrow [0,1]$ *such that*

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

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 $R_i : S_i \rightarrow \mathbb{R}$ such that*

$$R_i(s, \pi_i) = E_\tau \left\{ \sum_{t=0}^{\tau} \gamma^t R(s^\tau, \pi_i(s^t)) \,\middle|\, s^0 = s, \pi_i \right\}.$$

The discounted transition model defines for each state in the region S_i the probability of leaving the region through a certain exit state if following the policy π_i. These probabilities are discounted by the expected time until leaving the region. By the proof of the Composition Theorem (3.2.1) (see [30] for the proof), Sutton et al. showed that this discounting makes it possible to use these transition models in place of normal transition matrices in algorithms like value iteration.[6] Similarly, the discounted reward model specifies the expected discounted reward obtained when acting according to the policy starting in a certain state until termination, i.e. leaving the room. These models are in analogy to the state prediction matrix and the reward prediction vector of Sutton et al., except they are restricted in their predictions to the exit states, in accordance with the assumptions made about the termination condition β.

Generating Models

As mentioned, it is essential for planning to have appropriate models of the options we want to use. Thus, generating these models is a crucial step: For all $s \in S$, $s' \in XPer(S_i)$ the discounted transition probability for macro π_i satisfies:

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

[6] In fact, the discounting is used to guarantee a contraction mapping in the update formulas used in policy and value iteration, such that a unique solution can be ensured (compare [31] and the Banach-Fixed-Point Theorem).

By this, for each exit state s' a system of linear equations is defined, each of which containing $|S_i|$ equations with $|S_i|$ unknown variables. Note the difference between Tr and Tr_i: the former is the usual transition model of the global MDP, while the latter is the discounted transition model for macro π_i. Hence, they differ in the type of their second argument: Tr takes an action, recall that $\pi_i(s) \in A$, whereas Tr_i takes a policy as its second argument.

Similarly the expected discounted reward $R_i(s, \pi_i)$ for following π_i in state s satisfies:

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

Again a set of linear equations is defined. Solving these systems, either directly or by iterative methods, can be done in

$$\mathcal{O}(\underbrace{|XPer(S_i)| \cdot |S_i|^3}_{\text{transition probabilities}} + \underbrace{|S_i|^3}_{\text{exp. reward}}).$$

Creating Macros

So far, only the construction of models for *given* macros, i.e. policies, was discussed. However, the aim is to have options/macros generated automatically, that is, for a given region create some "good" policies. To judge the quality of macros one has to keep in mind their purpose: In the long run, we want to solve an MDP using these macros to save computational effort. Thus, a macro should be of use for this purpose. But when is a macro of use for solving an MDP? Of course, a macro can only be any help in the region of the MDP it is defined in. If the MDP is solved conventionally, not using macros, an optimal behavior (sub-policy) for this region would be computed. Hence, if there would be a (pre-computed) macro with exactly this policy, it would be of major help in solving the MDP, as it could simply be plugged in for this region. But, what does the policy depend on? In general, a policy entirely depends on the present values of all states. If the optimal value function is known for all states, the action to take in each state (policy at this state) can simply be choosen as the action leading to the adjacent state with the highest value, where adjacency is defined with regard to connecting actions. This similarly holds for regions: the policy for a region depends on the values of adjacent states, i.e. the exit states of the region, *plus* the values of all reachable states within the region. Therefore, if we knew the optimal value function for all these states, we could produce perfect macros. However, if we knew that function, the MDP would already have been solved and there would not be any need for macros. Also, the computational investment of creating macros does not pay off for solving one single MDP, and so what we are really after is to reuse such macros for various related MDPs. In these, the actual value function will certainly differ and thus the values the policy of the macro depends on should be general enough to keep the macro applicable to all these MDPs.

Before going on with this discussion, let us formalize how to obtain a macro for a given value assignment to the exit states of its respective region. This problem can be considered an, usually indefinite horizon-, problem which can be modeled as an MDP itself (taken from [21] into our notation):

Definition 3.2.2 *Let S_i be a region of MDP $M = \langle A, S, Tr, \mathcal{R} \rangle$ 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:*

(a) state space $S_i \cup XPer(S_i) \cup \{\alpha\}$ where α is a new reward-free absorbing state[7]*,*

(b) actions, dynamics, and rewards associated with S_i in M,

(c) a reward $\sigma(s)$ associated with each $s \in XPer(S_i)$,

(d) an extra single cost-free action applicable at each $s \in XPer(S_i)$ that leads with certainty to α.

Then the solution to this local MDP provides us with an optimal policy for this region given the values for the exit states.

To overcome the problems of not knowing the right value function for the exit states at the time a macro is computed, one could go about creating a large number of macros for different value functions, as discussed in [21]. However, this, in general, can get very expensive and usually unprofitable. Also, it would only make sense if the range of the value function is known.

Instead, heuristic approachs are considered, both by Hauskrecht et al. and Sutton and his group. One makes the assumption that the agent always wants to leave his current region via a certain exit. Thus, assigning a high positive value to only one exit and solving the corresponding local MDP, provides us with a policy for achieving this. This can be done for all exit states individually producing a set of macros.[8]

Abstract MDPs

Yet, the application of options/macros has only been discussed by intuition. One of the models of usage proposed in [21] is the following:

Definition 3.2.3 *Let $\Pi = \{S_1, \ldots, S_n\}$ be a decomposition of MDP $M = \langle A, S, Tr, \mathcal{R} \rangle$, 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 *$M' = \langle A', S', Tr', \mathcal{R}' \rangle$ induced by Π and $\mathcal{A}$, is given by:*

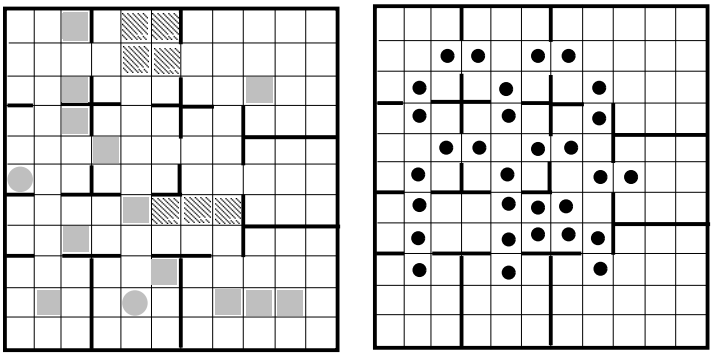
- *$S' = Per_\Pi(S) = \bigcup_{i \leq n} EPer(S_i)$*
- *$A' = \bigcup_i A_i$ with $\pi_i^k \in A_i$ feasible only at states $s \in EPer(S_i)$*
- *$T'(s, \pi_i^k, s')$ is given by the discounted transition model for π_i^k, for any $s \in EPer(S_i)$ and $s' \in XPer(S_i)$; $T'(s, \pi_i^k, s') = 0$ for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

This seizes our intuition about the general idea of options: We abstract from the original state space to a much smaller one, namely the set of peripheral states. These build some kind of interfaces between the regions. The actions used for planning are the options (macros) defined for the different regions. The models, transition model and reward function, are the discounted models that have been presented. Recall that these form the crucial part in planning at the level of options. We point out that this reduction of complexity, which will finally speed up computation as we will see, comes at the cost of possibly finding only a sub-optimal solution.

The examples in [21] have been conducted in a grid world which is depicted in Figure 3.6. This navigation task, where negative rewards are to be minimized, was solved

[7] Alternatively, one can define the local MDP without the additional state α and instead make all exit states absorbing with an empty feasible set.

[8] [21] also allows for the goal of staying in a region, modeled by low values for all exits.



(a) Maze121 (b) peripheral states

Figure 3.6: Taken from [21]: The example environment for testing the abstract MDP against the original MDP; (a) The agent can move in any compass direction or not move at all. Moving is uncertain: with some small probability the agent may move in another direction than intended. Each cell gives negative reward, except the upper right cell, which is absorbing and thus forms the goal. Shaded squares have a higher negative reward, on patterned squares moving is even more uncertain (probability of failure is higher). Shaded circles denote absorbing states with high negative reward; (b) peripheral states for a decomposition into 11 regions (rooms).

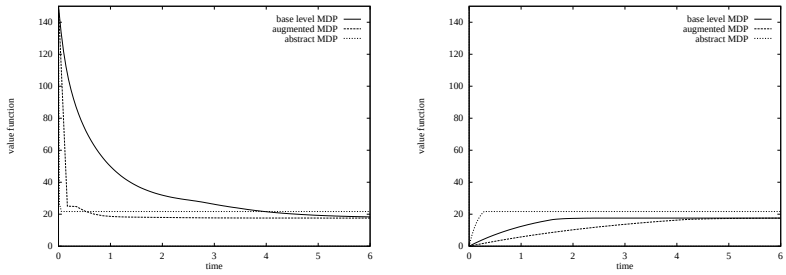
with the original MDP as well as with the abstract MDP. Additionally an *augmented* MDP was tested which we are going to leave out of consideration here. For the abstract MDP the set of macros was created based on the heuristic approach described above – one macro for each region-exit state combination, plus one for staying in the room.[9] Value iteration was applied to solve the different MDPs. Figure 3.7 shows the value for one particular state and how it improves over time. Clearly, with the abstract MDP the value function converges much faster. But, recognizable from the limit of the value function, the abstract MDP finds only a suboptimal solution: It finds a policy which takes the agent to the goal with expected costs (negative reward) of over 20, while the original MDP finds a way where less than 20 are expected. Nevertheless, the computational saving seem worth the drawback on solution quality.[10]

Hybrid MDPs

The main interest of Hauskrecht et al. is the reuse of macros which would justify the computational overhead of creating macros. To have a set of macros applicable to a set of related MDPs, these MDPs must bear sufficient similarities. In particular, it might happen that a goal moves within one specific region, leaving all other regions unchanged. This case is considered in [21]. To still be able to plan at the level of options, they argue to use a *hybrid MDP*. This kind of MDP is still composed by

[9] There are actually many ways of staying in a room. However, [21] does not provide any more detail on this question.

[10] Unfortunately, the speed up is only illustrated by figures in [21] and in particular no explicit numbers describing the speed up are given.



(a) (b)

Figure 3.7: Taken from [21]: Shown is the improvement of the value (negative reward) for one particular state over time during the value iteration process on the different MDPs; in case (a) the value for this state was initially overestimated, whereas in case (b) it was underestimated.

regions, but in those regions where changes are likely to happen the corresponding part of the original MDP is used, while in all other parts abstraction is applied.[11]

[11] To be precise, it would then seem better to merge all change-unlikely regions into one new region where abstraction is applied.

Chapter 4

Situation Calculus and Golog

The programming language GOLOG forms the basis for our considerations of the programming approach. In this chapter we will first present the situation calculus which GOLOG is based on. Later, after formally introducing GOLOG as it was initially proposed, we present those extensions of it which are relevant for our work, namely ICPGOLOG and DTGOLOG.

4.1 The Situation Calculus

The situation calculus is a second-order language that was first proposed by McCarthy [26]. The intention for this language was to represent and reason about dynamically changing worlds. The general idea was that a world only evolves due to the execution of *primitive actions* beginning in an *initial situation*. Three sorts are distinguished: *actions, situations* and normal *objects*. The initial situation is denoted by the constant S_0. This is the situation where yet no action has taken place. Further, the binary function symbol $do(a, s)$ denotes the successor situation after executing action a in situation s. *Fluents* represent properties of the world that change over time. We distinguish two types of fluents: *relational fluents* and *functional fluents*. For example, we could model the current student status of Bob as the relation $student(Bob, Now)$, where we use the special symbol Now to denote the current situation. A functional fluent $working\_hours(Bob, Now) = 70$, could state that Bob is currently working 70 hours a week.[1]

For each action there is an *action precondition axiom* stating the conditions under which the action can be executed. They can be represented in the form $Poss(a(\vec{x}), s) \equiv \Phi(\vec{x}, s)$, where $\vec{x}$ are the arguments of a. For example, the action $accept\_job(x, y)$ might have the precondition axiom

$$Poss(accept\_job(x, y), s) \equiv \neg student(x, s) \wedge job\_vacancy(y, s),$$

stating that x can only accept a job at y if and only if x is not a student and at y there is a job vacancy. Note that by this approach the qualification problem is simply ignored.

Further, the effects of an action have to be defined. This can happen via *effect axioms* describing the impact of actions on the world, i.e. the (truth-)values of fluents.

[1] Note that in our formulas symbols starting with an upper case letter denote constants, while variables start with a lower case letter. One exception is the special symbol Now which is always replaced by the current situation and in particular is not part of the language.

For a relational fluent F, for example, positive and negative such axioms can define the conditions under which the fluent is true ($\phi^+(\vec{x}, s)$), respectively false ($\phi^-(\vec{x}, s)$), after an action a is executed:

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

For example,

$$\begin{aligned} Poss(enroll(x, y), s) \wedge & \\ university(y, s) &\supset student(x, do(enroll(x, y), s)), \\ Poss(finish\_thesis(x), s) \wedge & \\ all\_exams\_passed(x, s) &\supset \neg student(x, do(finish\_thesis(x), s)), \end{aligned}$$

says that if person x can and does enroll at y and y is a university, then x will be a student (positive effect). On the other hand, if x is able to and does finish his masters thesis and he has passed all exams, then he will no longer be a student (negative effect).

While these axioms *do* describe the effects on certain fluents, they *do not* declare all the *non-effects* on other fluents. Axioms describing these are called *frame axioms*. The *frame problem* expresses the impossibility of stating and reasoning with all frame axioms, i.e., all the non-effects of actions. There are far to many. Even apparently ridiculous things, like "finishing one's thesis does not change one's gender" would have to be captured by a frame axiom:

$$gender(x, s) = y \supset gender(x, do(finish\_thesis(x), s)) = y.$$

4.1.1 A Solution to the Frame Problem for Deterministic Actions

Ray Reiter [33] proposed a solution to the frame problem based on a completeness assumption: For each fluent the impact of all (deterministic) actions[2] on it are collected, that is, all the effect axioms mentioning the fluent in question. From that, syntactically one can generate a *successor state axiom* for the fluent which states the known ways the fluent may change. The core of Reiters approach is to assume that these successor state axioms are complete in that they list *all* possible ways by which the fluent can change. The following would then be the successor state axiom for the fluent $student(x, s)$:

$$\begin{aligned} Poss(a, s) \supset [student(x, do(a, s)) \equiv (a = enroll(x, y) \wedge university(y, s)) \\ \vee (student(x, s) \wedge \neg(a = finish\_thesis(x) \wedge all\_exams\_passed(x, s)))] \end{aligned}$$

In the following we describe how Reiters solution applies to *functional fluents*[3], for relational fluents the computations are similar and can be found in [33]. The effect axiom for a functional fluent f and action A have the form ($\vec{t}$ are terms):

$$Poss(A, s) \wedge \phi_f(\vec{t}, y, s) \supset f(\vec{t}, do(A, s)) = y$$

[2] This, only valid for deterministic actions, is the restriction that brought forth the addition "sometimes" in [33].

[3] We chose to present the computation in detail for functional fluents, because it is the theoretical basis for the automatic transformation from effect axioms to successor state axioms done by the preprocessor of Section 5.3.

Note that for functional fluents there are no positive and negative effect axioms like for relation fluents, but only one axiom explicitly stating the new value (y) of the fluent. Above formula can be rewritten to:

$$Poss(a,s) \wedge \underbrace{a = A \wedge \vec{x} = \vec{t} \wedge \phi_f(\vec{x}, y, s)}_{\Phi_f} \supset f(\vec{x}, do(a,s)) = y$$

which can be done for all n effect axioms for fluent f. All these can then be joint into a single *normal form* for the effect axiom:

$$\begin{array}{rcll} Poss(a,s) \wedge [\Phi_f^{(1)} \vee \ldots \vee \Phi_f^{(n)}] & \supset & f(\vec{x}, do(a,s)) = y, & \text{or} \\ Poss(a,s) \wedge \gamma_f(\vec{x}, y, a, s) & \supset & f(\vec{x}, do(a,s)) = y & \end{array} \quad (4.1)$$

The completeness assumption then expresses that if fluent f changes its value from situation s to situation $do(a,s)$, then $\phi_f(\vec{x}, y, a, s)$ must be true:

$$Poss(a,s) \wedge f(\vec{x}, s) = y' \wedge f(\vec{x}, do(a,s)) = y \wedge y \neq y' \quad \supset \quad \gamma_f(\vec{x}, y, a, s) \quad (4.2)$$

Together with the assumption

$$\neg \exists \vec{x}, y, y', a, s. Poss(a,s) \wedge \gamma_f(\vec{x}, y, a, s) \wedge \gamma_f(\vec{x}, y', a, s) \wedge y \neq y'$$

Reiter shows that (4.1) with (4.2) is logically equivalent to:

$$\begin{array}{rcl} Poss(a,s) & \supset & [f(\vec{x}, do(a,s)) = y \equiv \gamma_f(\vec{x}, y, a, s) \vee \\ & & f(\vec{x}, s) = y \wedge \nexists y'. \gamma_f(\vec{x}, y', a, s) \wedge y \neq y'] \end{array} \quad (4.3)$$

which is called the *successor state axiom for functional fluent* f ([34]).

4.1.2 Basic Action Theory

Levesque et al. [24] propose to formulate a *basic action theory* $\mathcal{D}$ to describe the world and its dynamics:

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

with

- Σ the set of (domain independent) foundational axioms for situations (e.g. $S_0 \neq do(a,s)$);
- $\mathcal{D}_{ss}$ a set of successor state axioms, one for each fluent;
- $\mathcal{D}_{ap}$ a set of action precondition axioms, one for each action a;
- $\mathcal{D}_{una}$ a set of unique name axioms for actions;
- $\mathcal{D}_{S_0}$ a set of axioms describing the world in the initial situation S_0.

To illustrate how to formulate such a theory, we lay it out with our student example:

$$\begin{array}{ll} \mathcal{D}_{S_0} = \{ & student(Bob, S_0), \\ & all\_exams\_passed(Bob, S_0), \\ & gender(Bob, S_0) = Male, \end{array}$$

$$
\begin{aligned}
& working\_hours(Bob, S_0) = 70, \\
& university(RWTH - Aachen, S_0), \\
& job\_vacancy(Porsche, S_0)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D}_{ap} = \{ \quad & Poss(accept\_job(x, y), s) \equiv \neg student(x, s) \wedge job\_vacancy(y, s), \\
& Poss(enroll(x, y), s) \equiv TRUE, \\
& Poss(finish\_thesis(x), s) \equiv student(x, s)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D}_{ss} = \{ \\
& [student(x, do(a, s)) \equiv (\exists y.a = enroll(x, y) \wedge university(y, s)) \\
& \vee student(x, s) \wedge \neg(a = finish\_thesis(x) \\
& \wedge all\_exams\_passed(x, s))], \\
& [all\_exams\_passed(x, do(a, s)) \equiv FALSE \vee \\
& all\_exams\_passed(x, s) \wedge TRUE], \\
& [gender(x, do(a, s)) = y \equiv FALSE \vee \\
& gender(x, s) = y \wedge TRUE], \\
& [working\_hours(x, do(a, s)) = y \equiv \\
& (\exists z.a = accept\_job(x, z) \wedge y = 38.5) \vee \\
& (\exists z'.a = enroll(x, z') \wedge university(z', s) \wedge y = 70) \vee \\
& (working\_hours(x, s) = y \wedge \\
& \nexists y'.(((\exists z.a = accept\_job(x, z) \wedge y' = 38.5) \vee \\
& (\exists z'.a = enroll(x, z') \wedge university(z', s) \wedge y' = 70)) \wedge y \neq y'))], \\
& [university(x, do(a, s)) \equiv FALSE \vee \\
& university(x, s) \wedge TRUE], \\
& [job\_vacancy(x, do(a, s)) \equiv FALSE \vee \\
& job\_vacancy(x, s) \wedge TRUE] \qquad \}
\end{aligned}
$$

The successor state axioms containing $TRUE$ and $FALSE$ are cases where no action which affects the respective fluent exists. Consequently, there is no condition under which the fluent becomes true or its value is changed. However, there neither is any condition making it false. Thus, the fluent will always keep its original (truth-)value as defined in $\mathcal{D}_{S_0}$. Here we left out the domain independent foundational axioms Σ and the unique name axioms $\mathcal{D}_{una}$ which are straight forward to formulate.

Using the basic action theory we can derive the value of any fluent in the current situation by what is called *regression*. Roughly, regression for a given fluent f and situation $do(a_n, do(\ldots do(a_1, S_0)\ldots)$ works like this: Applying the successor state axiom of f once, will describe the current value of f possibly relative to its value in the previous situation. If the value does not depend on the previous situation, we are done. Otherwise, the value in the previous situation can again be computed by applying the successor state axiom to that situation. This way, recursively the regression will eventually end up in situation S_0, where the value of fluent f can be determined from the axiomatization of S_0.

Naturally, the described algorithm takes more time the longer the situation term gets (see Section 5.3 for a quantitative analysis). This can be a problem for realistic

domains, especially for those where the agent has a nonterminating program to run. Fortunately, there has been approaches to circumvent this problem. In Section 4.2.1 one of these approaches is briefly described.

4.2 Golog

The action programming language GOLOG [25] is based on the situation calculus. It can use it to *project* how the world would evolve if a certain sequence of primitive actions was chosen. GOLOG offers the following common programming constructs to formulate complex actions over primitive ones:

- nil the empty program;
- a, primitive actions ($\hat{=}$ actions from the situation calculus);
- $[e_1, \ldots, e_n]$, sequences;
- $?(c)$, tests: if condition c is true proceed
- $if(c, e_1, e_2)$, conditionals: if condition c is true proceed with sub-program e_1 else proceed with sub-program e_2;
- $while(c, e)$, loops: while condition c is true repeat sub-program e;
- $e_1|e_2$, nondeterministic choices: do e_1 or e_2;
- $star(e)$, nondeterministic repetitions: repeat sub-program e an arbitrary number of times;
- $pi(v, e)$, nondeterministic choices of argument v: choose an arbitrary term t and proceed with e, where all occurrences of v are substituted by t;
- procedures;

where the e_i are legal GOLOG programs.

One way of defining the semantics of these constructs is by an *evaluation semantics* like originally used in [25]: Formally the above statements are abbreviations for formulas in the situation calculus. Their translation into formulas is defined via the predicate $Do(\delta, s, s')$ which states that executing program δ in situation s will result in the new situation s'. Thus, an evaluation semantics is a set of definitions like:

$$Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$$

where $a[s]$ is the action a with all fluents among the arguments evaluated in situation s.[4] This defines that an action a on execution in situation s will end in the new situation $do(a[s], s)$ if $a[s]$ is possible in s. Likewise, for all other constructs δ, $Do(\delta, s, s')$ is defined. Conditions are also evaluated in the actual situation:

$$Do(if(c, e_1, e_2), s, s') \stackrel{def}{=} c[s] \wedge Do(e_1, s, s') \vee \neg c[s] \wedge Do(e_2, s, s').$$

[4] Formally, $a[s]$ states that all fluents appearing as an argument to a get s set as their actual situation argument.

Nondeterminism is simple defined by a disjunction:

$$Do((e_1|e_2), s, s') \stackrel{def}{=} Do(e_1, s, s') \vee Do(e_2, s, s').$$

For an arbitrary GOLOG program Δ and a situation s a constructive proof of $Do(\Delta, s, s')$ would return a situation statement in s' of the form $do(a_n, do(a_{n-1}, \ldots do(a_1, s)) \ldots)$. This implicitly contains the action sequence $a_1, \ldots, a_n$ which is a possible course of action to get from situation s to situation s'. In a nutshell, what GOLOG does is to find such a constructive proof for any input program p and situation s, where the instantiated new situation s' is considered the result.

To illustrate the original intention for nondeterminism, consider the following legal program:

$$E = [a, (b|a), ?(\phi)].$$

Let a and b be primitive actions and ϕ a condition. Suppose ϕ is initially true and action a toggles its truth-value each time a is executed and b does not affect ϕ. Then the only proof for $Do(E, S_0, s')$ will return $s' = do(a, do(a, S_0))$ as this is the only way of making ϕ true after already having executed a at the beginning. Thus, the nondeterminism is resolved as to proof the entire program. Note that if in the above case both alternatives would have made ϕ true, there would be two possible proofs and there would be no decision rule stating preference for a or b.

When looking for an action sequence to achieve some kind of goal, for example formulated as a condition, GOLOG can be used to restrict the search space: instead of always only nondeterministically choosing among the primitive actions, the control constructs can be used to provide some form of plan skeleton. This can be used to reduce branching and thus computational effort in finding a (linear) *plan* for achieving the goal.

In fact, GOLOG has successfully been used to control the museum tour guide robot Rhino ([8]) at a museum in the city of Bonn, Germany.

4.2.1 icpGolog

Unfortunately, the expressiveness of the original GOLOG was not strong enough to model some of the properties of realistic domains, especially mobile robotics, which from the beginning was one of the main application domains for GOLOG. Consequently, many extensions of GOLOG have been proposed such as [13, 12, 23, 19, 20]. Many of these extensions have recently been merged into a new derivative called ICP-GOLOG and an interpreter has been implemented in ECLiPSe Prolog [10].

The following previously proposed ideas and extensions of GOLOG have been incorporated into ICPGOLOG:

on-line : The ICPGOLOG interpreter is an *on-line interpreter*. The characteristic of such an interpreter is that programs are executed right during interpretation (on-line). The interpreter works *incrementally*, that is, after the interpreter did a step in the program it commits to it by executing the respective action in the real world. This fashion of interpretation was first proposed by Levesque and De Giacomo [12] and was required to avoid delays in the execution of long programs containing nondeterministic choices. Imagine a long program were at the beginning a nondeterministic choice is to be made and assume that this choice influences a test at the end of the program. Then, to guarantee the successfull

termination of the entire program, it would be necessary to project the program to the end, possibly for both choices, before making a decision. The execution of the first action cannot wait until program interpretation terminates. It even happens that programs are non-terminating, assigning the robot a job to do continuously.

Also, an incremental proceeding naturally supports *sensing*[5]: Certain properties of the world may initially be unknown[6], but can be sensed during execution using *sensors*. We understand sensors as any kind of mechanism that can provide the agent with information of any kind which is currently true in the world. In robotics, sensors are typically laser range finders, cameras, sonars, microphones or bumpers, but can also be more abstract like direct user input or a mechanism to query websites with current stock information. The need for sensing becomes obvious in the following example: Consider the task of catching a plane at the airport. You may know how to get to the airport and how to reach a certain gate. But what you usually do not know in advance, that is before reaching the airport, is the particular gate the plane is leaving. Thus, this information has to be sensed when the execution of the plan has advanced to the moment where you reach the airport.

Additionally, the interpreter takes so-called *exogenous actions* into account. Exogenous actions are actions that are beyond the control of the agent. The agent can neither execute them, nor can it in general predict their occurrence. Nonetheless, these actions change the world and therefore the values of fluents. The interpreter supports these actions as long as their effects are known. Then, whenever such an action happens, the interpreter changes the values of the fluents according to the described effects. Exogenous actions can be compared to sensing actions as both provide the agent with information about the world. From that point of view, the difference lies in the conditions under which the information is retrieved: sensing actions can be seen as polling, while exogenous actions resemble an interrupt.

continuous change : Grosskreutz and Lakemeyer first proposed a notion for representing continuously changing properties of the world [19]. This extension is directly added to the situation calculus and has mainly the following ingredients:

- a new sort *Real* ranging over the real numbers;
- a special functional fluent $start$ with the intuition that $start(s)$ denotes the starting time of situation s;
- a new sort *t-function* representing functions of time;
- a new binary function val to evaluate a t-function at a given time;
- a new type of *continuous fluents* whose values are functions of time, i.e. t-functions;
- a new action $waitFor(\tau)$ to advance the time, i.e., increase the value of fluent $start$, in off-line mode to the least time point when the condition τ holds.

Then we can model things like a continuously, 1-dimensionally moving robot like this [18]: We create a new continuous fluent $robotPos$ to denote the robots

[5]For a discussion on sensing and off-line interpreting see [23].

[6]This could, for example, be modeled by an incomplete axiomatization of the initial situation S_0.

position over time and a t-function $linear(x_0, v, t_0)$ whose value at time t is defined by $val(linear(x_0, v, t_0), t) = x_0 + v \cdot (t - t_0)$. Then we assign the fluent $robotPos$ the value $robotPos(S_0) = linear(4.3, 1.0, 0.0)$ stating that initially at time 0.0 the robot is at position 4.3 and moves with velocity 1.0. If the time is now advanced to $t' = 2.0$, off-line by performing an adequate $waitFor(\tau)$ action or on-line by some externally determined passage of time, the value of $robotPos$ can be evaluated at the current time to $val(linear(4.3, 1.0, 0.0), 2.0) = 4.3 + 2.0 \cdot (2.0 - 0.0) = 6.3$. For further details we refer to [18].

concurrency : ICPGOLOG has taken over the concept of concurrency like it was first proposed for the CONGOLOG interpreter [13]. Concurrency is understood as interleaving two programs and in particular does not consider actions being truly simultaneous. This avoids problems like the precondition interaction problem (see [28]) and the otherwise necessary extension of the underlying situation calculus. However, the semantics of concurrency in ICPGOLOG is taken over from Grosskreutz CCGOLOG [18] and differs from the semantics in CONGOLOG: concurrency of two programs σ_1 and σ_2 is expressed by the construct $conc(\sigma_1, \sigma_2)$. The general assumption is that actions should perform as *soon* as possible. Thus, the next construct of both programs is considered. If both can do an instantaneous transition σ_1 is favored (compare to prioritized concurrency of CONGOLOG). However, if one of the two transitions takes longer than the other, the earlier terminating transition is favored. This kind of time dependent selection was not possible in CONGOLOG, because it did not contain any concept of time.

probabilism : In realistic domains uncertainty exists in various forms, one of which is uncertainty about how the world evolves. This often can be modeled by a list of possible outcomes together with a distribution defining their respective probabilities, for example based on experience. This way of modeling uncertainty gave birth to PGOLOG [20]. PGOLOG introduces a new construct $prob(p, \sigma_1, \sigma_2)$ with the intuition that with probability p sub-program σ_1 gets "executed" and with remaining probability $(1 - p)$ σ_2 is chosen. However, this kind of construct is not intended for execution, but only for models used in projection: PGOLOG offers a mechanism for *probabilistic projection*. The user can, given a legal PGOLOG-program, query the probability that a certain condition is true after executing that program. This in turn can be used for decision making.

Roughly, the following is suggested: Along with the program itself the user provides a model for the low-level processes, like navigation in robotics. Then, using concurrency as presented above and a special architecture for communicating with the low-level processes[7], concurrently interpreting the low-level model and the program projects the expectations about the effects of this program.

progression : Following the approach for progressing a database by Lin and Reiter [15], Jansen [22] implemented a mechanism to progress the knowledge base in ICPGOLOG according to the current situation. The motivation for progression is simple: once an action has been performed in the real world, it cannot usually be undone. Thus, situations before performing the action render irrelevant. However, in realistic domains the situation term often grows rapidly over time,

[7] Confer [18] for details.

requiring more and more space (memory) and computational time to perform regression. Hence, it is suggestive to *progress* the knowledge about what was true in situation S_0 to what is true in the current situation.

It turns out that this feature is indispensable for realistic domains, for the arguments that motivated it (time and space consumption).

Transition Semantics

Different from the original GOLOG, the semantics of ICPGOLOG is defined via a *transition semantics*. The evaluation semantics of GOLOG was first replaced by the transition semantics in [13] when introducing concurrency. The $Do(\delta, s, s')$ predicate of GOLOG assigns a semantics to the entire program δ recursively. This semantics is based on the complete evaluation of the program, therefore its name. However, sometimes it is more convenient to specify the semantics by defining single computational steps, which is the idea of a transition semantics. The semantics is defined via axioms of a new four-ary relation $Trans$. $Trans(\delta, s, \delta', s')$ holds if and only if one execution step of program δ in situation s leads to situation s' and remaining program δ'. A tuple $\langle \delta, s \rangle$ with δ a program and s a situation is called *configuration*. Thus, $Trans$ defines transitions from one configuration to another.[8]

Moreover, we have to define what a termination configuration is, i.e., what are configurations that we consider a successful termination of a program. This is done by another predicate $Final$. $Final(\delta, s)$ holds if and only if we consider the configuration $\langle \delta, s \rangle$ a legal termination. Here are some examples for $Trans$- and $Final$-definitions (for a complete list see Appendix A.2):

$$\begin{aligned}
Final(a, s) &\equiv FALSE \quad \text{, where } a \text{ is a primitive action} \\
Final(if(\phi, \sigma_1, \sigma_2), s) &\equiv \phi[s] \wedge Final(\sigma_1, s) \vee \neg\phi[s] \wedge Final(\sigma_2, s) \\
Trans(a, s, nil, s') &\equiv Poss(a[s], s) \wedge s' = do(a[s], s) \\
Trans([\sigma_1, \sigma_2], s, \delta, s') &\equiv \exists\gamma[Trans(\sigma_1, s, \gamma, s') \wedge \delta = [\gamma, \sigma_2] \vee \\
&\quad\; Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s')
\end{aligned}$$

For modeling probabilism in PGOLOG the transition relation $Trans$ was formally converted to a function $transPr$ which returns the probability for the given transition. However, this is usually equal to 1.0 except for $prob$ statements. For example:

$$\begin{aligned}
&transPr(a, s, \delta, s') = q \equiv \\
&\qquad Poss(a[s], s) \wedge \delta = nil \wedge s' = do(a[s], s) \wedge q = 1.0 \vee \\
&\qquad \neg(Poss(a[s], s) \wedge \delta = nil \wedge s' = do(a[s], s)) \wedge q = 0.0 \\
&transPr(prob(p, \sigma_1, \sigma_2), s, \delta, s') = q \equiv \\
&\qquad \delta = \sigma_1 \wedge s' = do(tossHead, s) \wedge q = p \vee \\
&\qquad \delta = \sigma_2 \wedge s' = do(tossTail, s) \wedge q = 1 - p
\end{aligned}$$

Here, $tossHead$ and $tossTail$ are pseudo-actions introduced to make the resulting situations different. They have no effect on any fluent and are always possible.

[8] It is noteworthy that a transition semantics requires the reification of programs as first-order terms in the logical language. This is basically because one needs to quantify over programs. For the Do definitions that was not necessary, since these we simply (recursive) abbreviations for formulas over s and s'. However, we omit any detail of encoding programs as first-order terms, as it does not affect our issues, and refer to the literature [13, 18].

Decision Making

Probabilistic projection can be used for decision making. Jansen [22] shows examples for this in the RoboCup Simulation League: Suppose the agent has the ball under its control, wants to play a pass and is considering possible pass receivers. Then it can use a model for the low-level action "pass" to project the possible outcomes for passing to a certain teammate and query the probability for certain properties in these outcomes. Jansen suggests to use a condition stating that the teammate successfully received the pass: The agent queried the probability for each possible pass to succeed. This was iteratively done for some ordering of some teammates. The first one which had a probability greater than some threshold was chosen.

However, this does not seem to be a sophisticated decision rule for a couple of reasons: (a) the decision depends on the ordering of the teammates, (b) usually not the best player will be selected, (c) if for no player the threshold is reached, the pass playing agent has no clue what to do. What essentially is missing, is the support for a decision-theoretic view on such decision, like in MDPs. This should be directly implemented into the interpreter. While the above procedure works with a hand-coded decision tree using if-then-else constructs, it seems suggestive to use nondeterministic constructs to model the decision problem when the interpreter itself is capable of decision making.

icpGolog and Nondeterminism

However, icpGOLOG does not contain nondeterministic instructions anymore. Grosskreutz [18] explains the problems arising from the interplay of nondeterminism and his semantics of concurrency. This lead to the complete omission of nondeterminism in ccGOLOG, which was later taken over into icpGOLOG. The arguments he gave do not prevent reintroducing nondeterminism in general. Grosskreutz was only concerned about *counterintuitive results* (see [18] for an example) that can only occur when mixing concurrency and nondeterminism. His claim is that in such cases nondeterministic choices are, under certain circumstances, determined according to the least time consumption. He believed that to be in contradiction to the intuition behind nondeterminism. Yet, when reintroducing nondeterminism we are going to prohibit the use of concurrency with nondeterminism for similar reasons (see Section 5.1.3).

4.2.2 DTGolog

In [5] Boutilier et al. propose another GOLOG extension which they call DTGolog (*decision-theoretic* GOLOG). Roughly, DTGolog integrates fully-observable MDPs into GOLOG and this way allows combining explicit programming with decision theoretic planning.

MDP specification

Boutilier et al. do not specify the MDP formally, but the following describes how the components of the MDP could be defined:

state space = situations : The set of states is the set of situations. Unfortunately, in general the set of situations is infinite. To illustrate this, consider the vacuum world of Figure 4.1. The vacuum cleaner can move freely between the two rooms and in both rooms there is either dirt or they are clean. Then we can represent the state space as the crossproduct of the three variables for the position of the

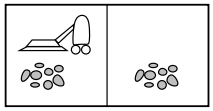


Figure 4.1: The vacuum world taken from [35].

robot, the presence of dirt in the left room, and the presence of dirt in the right room. Such a state space is called *factored* (cf. [6]), as it is specified using more than one state variable. The size of this particular state space is $2 \cdot 2 \cdot 2 = 8$ which is simply the number of all possible combinations of values for the three variable. However, in DTGOLOG the state space for this example is infinite: assume there is only one action $changeroom$, which is always possible. Then, after performing this action twice, the vacuum cleaner ends up in the room he departed from. Intuitively, and according to our state description above, it is in the same state as before. But this is not true for the situation: if the situation initially was S_0, it will now be $do(changeroom, do(changeroom, S_0))$ which of course is different. One approach to this rather undesired effect, is to specify a list of fluents that are used as state variables. We will apply this approach in Section 5.2 when constructing a finite state space for options.

action space = primitive actions : The actions of the MDP can simply be understood to be the primitive actions in GOLOG. However, the situation calculus does not allow for uncertain effects of actions (recall that Reiters solution to the frame problem explicitly requires deterministic actions). Thus, to still be able to model uncertainty, Boutilier et al. added a new type $agentAction(\mathbf{a})$[9] and a new predicate $nondetActions(\mathbf{a}, s, \Omega)$ indicating that agent action $\mathbf{a}$ in situation s is nondeterministic and has the outcomes which are listed in Ω, commonly called *nature's choices*. This list of outcomes consists of primitive actions of the situation calculus. If an agent action is not nondeterministic, it is immediately a primitive action of the situation calculus. Additionally, the new predicate $prob(a, p, s)$ states the probability p that primitive action a happens, if in situation s a nondeterministic action is executed with a being one of its outcomes. This approach is similar to Reiter's stochastic GOLOG [34].

In DTGOLOG, fully-observable MDPs are considered. Thus, it has to be guaranteed that the agent always knows the situation (state) it is. However, the current situation is not individually sensible from the world. To illustrate this, we once more consider the vacuum world: Assume the robot is in the left room and it wants to determine its situation. The only information perceivable from the world, assuming required sensors are available, are the values of the three state variables (robot position, dirt left, dirt right). However, as pointed out, there are several (in fact infinitely many) situations that correspond to a certain value configuration. Or in short: the mapping from states to situations is not bijective. Nevertheless, if the robot keeps track with its moves it can always memorize its situation. For deterministic actions this is simple. For the newly added nondeterministic actions, however, extra care is needed: after executing such an action, the agent only knows which situation calculus actions *may* actually have happened. But it still has to determine which exactly it was. For each agent action

[9] We are using bold face for agent actions ($\mathbf{a}$). All other actions are primitive actions.

$\mathbf{a}$, the user therefore specifies a *sensing action senseEffect*($\mathbf{a}$) which performs all necessary sensing to distinguish the alternatives. Further, with every possible outcome of action $\mathbf{a}$, he assigns a *sense condition* which holds if and only if this outcome has actually happened.

Concluding, full observability is assured by keeping track of what the agent is doing. While for deterministic action the effects are certain, the agent uses particular sensing actions and conditions for determining the outcome for nondeterministic actions.

transition function = SSAs: The transition function is trivially defined by the successor state axioms and the new predicates $nondetActions(\mathbf{a}, s, \Omega)$ and $prob(a, p, s)$. With these we can always tell the probability of reaching a certain situation when applying a specific agent action $\mathbf{a}$.

reward function = reward(s, r): Another new predicate $reward(r, s)$ specifies the reward r assigned to situation (state) s. Since s is a situation term, it contains the entire history and in particular the last action that has taken place. It is therefore possible not only to judge the situation itself, but also to subtract costs based on the last action.

A New Semantics for Nondeterminism

DTGOLOG, like GOLOG, has an evaluation semantics, defined by the predicate $bestDo(\delta, s, h, p, v, t)$ with:

- δ a program (as before),
- s a situation (as before),
- h a natural number called the *horizon*,
- p the optimal *policy* for the next h steps in program δ when starting in situation s,
- v the expected *value* for this policy,
- t the *termination probability* in the usual GOLOG sense for program δ when following this policy.

Compared to $Do(\delta, s, s')$ of GOLOG, the resulting situation s' is missing. Instead a policy is *returned*, reflecting the uncertainty of the system.[10] This policy is a GOLOG program only containing primitive actions, conditionals and tests. Since we are in the special case of knowing where the agent starts, namely in situation s, we can represent a policy as a conditional program, containing primitive actions and if-then-else statements to account for the uncertainty of some of the actions. The policy implements the mechanism for keeping up full-observability as described above.

The horizon h can be used to restrict the interpretation up to a certain number of actions. Then, the interpretation returns either when reaching a termination situation, just like in GOLOG, or when the horizon is reached.

[10] We will speak frequently of arguments being *returned* or distinguish between *input* and *output* arguments. Although from a logical point of view there is no such type distinction, as all are only arguments of a relation, this *functional* view on predicates is rather intuitive and should be helpful for understanding the flow of control in the interpreter.

Most of the Do clauses translate trivially to $bestDo$. The following are the most relevant differences:

nondeterministic choice (user's choice):

$$bestDo([(\sigma_1 | \sigma_2), \sigma], s, h, p, v, t)$$

The interpreter recursively calls $bestDo$ for both alternatives σ_1 and σ_2. Based on the returned values and termination probabilities, it chooses the more promising, adds it to the policy, and returns its values. Thus, unlike GOLOG, where nondeterminism was a simple disjunction, here the *better* alternative is used.

nondeterministic action (natures choice):

$$bestDo([\alpha, \sigma], s, h, p, v, t)$$

For an agent action α, it is tested whether this action is deterministic or not. Since the former case is equally treated as in GOLOG, we here only describe the latter. In that case, the list of outcomes Ω for this action is retrieved from $nondetActions(\alpha, s, \Omega)$. As in the user's choice case, for each entry of the list, the remaining projection is run, i.e. $bestDo$ is called. However, unlike the former case, the interpreter cannot *choose* the best alternative. It is not in the hand of the agent to decide, but nature's choice. Accounting for that, the *average* of all alternatives is returned. For value and termination probability this means that the results of the alternatives are added where each alternative o is weighted by its probability of occurrence, as defined by $prob(o, \alpha, s)$. The policy is constructed from the sensing action *senseEffect*(α) and if-then-else constructs were the sense conditions are used to detect the actual outcome.

This way of solving the problem is obviously different from the iterative algorithms described in Section 3.1.5: The main difference is that the algorithm at hand solves the problem only for a particular starting point, the current situation, instead of iterating over the entire state space. The algorithm is known as *decision tree search* [6] or *directed value iteration* [5], as the resulting policy is a decision tree rooted in the initial situation. This is in contrast to the definition of a policy of Section 3.1.4 where a policy was defined as a mapping from the set of states to actions. The great advantage of the algorithm is that it can operate even on infinite state spaces. This is trivially true, because by construction, the algorithm will only reach a finite number of states during processing. This part of state space is explored as some kind of reachability graph from the initial situation.

The disadvantage of this algorithm is the following: As described above, the algorithm takes situations as states. Depending on the properties of the domain, it can often happen that intuitively equal states are treated as different. This means that for all these, the remaining projection is run, although it is always the same. In that case, computational complexity is increased unnecessarily.[11]

From these properties it follows that this algorithm is especially useful for domains with a continuous state space. There the described advantage has its effect, while the disadvantage does usually not, since in continuous state spaces it is unlikely to pass through a certain state more than once in a decision tree search.

[11] We present a solution to this problem in the Section 5.2.

The central point of DTGOLOG is that nondeterminism can freely be combined with common programming constructs. Seen from the perspectives where the components (decision-theoretic planning and programming) originate, this can been understood differently:

- Seen from the point of view of programming, DTGOLOG simply adds nondeterminism with a decision-theoretic semantics to the list of allowed constructs.
- From the standpoint of decision-theoretic planning, DTGOLOG allows to restrain the search space by incorporating domain knowledge, represented by deterministic decision rules. These deterministic decision rules are formulated in terms of programming constructs.

An Example

To illustrate how in practice the combination of programming and planning can look like, we present three different approaches to a simple navigation problem in the Maze66 (cf. Figure 2.1 (a)). Starting in cell $(1, 1)$ the agent can move in any compass direction, where a move succeeds as intended with probability 0.91 and otherwise leads to any other adjacent cell. Each move costs 1 and at position $(4, 4)$ there is a high positive reward.

1. A simple programming approach to this problem could be the following:
   ```
   [while( y < 4, down), while(x < 4, right)]
   ```
   which first goes down to the correct y coordinate, and then heads right.

2. A planning approach could leave all decision to the agent:
   ```
   while( not(x = 4 \& y = 4), (down | right | up | left))).
   ```
   That is, while the position is not $(4, 4)$, choose nondeterministically to go in any possible direction.

3. Finally an integrated approach could be:
   ```
   ( [while( y < 4, down), while(x < 4, right)]
   | [while( x < 4, right), while(y < 4, down)] )
   ```
   which nondeterministically chooses one of two ways to take to the goal.

All these approaches will find the best path to the goal cell. However, they differ in quality: if we use the required computational time and the flexibility of the approaches as quality criteria, the following picture is drawn: The programming approach takes almost no time, but it is only applicable for exactly this problem and for deterministic actions. If one of the two doors the robot has to pass through were closed, the program would no longer reach the goal. Also if one of the used actions fails, the program will loose its track and fail. The second approach is highly flexible and can, as long as the uncertainty of the moves is known, handle nondeterministic actions. It will find a solution for any position of the goal if one exists. However, it is computational expensive and, in particular when using the decision tree search algorithm of DTGOLOG, takes prohibitively long. The third approach can be seen as the result of incorporating the knowledge that one of the doors along the two possible shortest ways may be closed. On the downside it neither supports nondeterministic actions. The somewhat higher flexibility of the approach is paid by a little bit longer computation.

The example illustrates that using DTGOLOG it is possible to incorporate domain knowledge to increase flexibility, by integrating planning with programming. It is up to the programmer to decide on how much uncertainty to account for, by using more nondeterminism in the program. This enables to react upon surprising changes in the world, like a closed door in this case. Additionally it is noteworthy that even without uncertainty in the domain, a planning approach is generally easier to implement and finds the best possible solution if the world is correctly modeled.

4.2.3 Online DTGolog

In the introduction we sketched the shortcomings with DTGOLOG. One of which were that the DTGOLOG interpreter is a pure off-line interpreter. In analogy to the motivation for an incremental GOLOG interpreter described in [12], Soutchanski [37] recently proposed an on-line extension of DTGOLOG. He recognizes the need for an incremental interpreter that allows for sensing actions. As for the incremental GOLOG interpreter, he claims that solving a program to the end before executing a first action takes too long and may even be unnecessary if the program is a sequence of logically independent problems that can be solved one after another.

The solution Soutchanski suggests is that of adding another argument to $bestDo$ stating the remaining program after the first step of the policy has been executed. The general execution semantics of the new interpreter is as follows: In each step, the remaining program is interpreted off-line up to a given horizon constructing a policy, just like in DTGOLOG. Then the first action of the policy is executed and the process proceeds with the remaining program. This way, at *each* step a projection is performed. This is even the case, if there is no nondeterminism used within the program up to the given horizon. To better control the search, Soutchanski introduces two new constructs $optimize(\sigma)$ and $local(\sigma)$. The former can be used in sequences to constrain the search to the program σ ignoring any remaining program. If $local(\sigma)$ is used, a policy for σ is constructed (up to the given horizon) which then replaces the original program σ.[12] That is, the interpreter commits to this policy without having any second thought while execution. Although this intuitively should allow to execute a larger sequence of action without much computational expenses, this is only partially true: As the general semantics of the interpreter always optimizes over the next steps up to the given horizon, even after using $local$ the interpreter will try to optimize the already deterministic program which is the policy returned by $local$. Although projecting a deterministic program is usually much faster than projecting a program containing nondeterminism, it is completely unnecessary.

As a result for our work, it remains to say that Soutchanski's work addressed the right problems, but still appears to leave space for improvements. Our approach to the issue of on-line decision-theoretic planning, which we will address in Section 5.1, will follow the ideas of the incremental GOLOG of [12] for nondeterminism and using a transition semantics. In particular, we not generally optimize a program before execution. Instead we will only search for a policy when requested by the user through a search operator similar the one proposed in [12].

[12] Unfortunately, [37] does not explain how $local(\sigma)$ behaves in case the horizon does not suffice to render a policy for the entire program σ. If in that case still σ were replaced by the policy, the part of σ behind the horizon would be discarded, which obviously cannot be intended.

Chapter 5

ReadyLog

Based on ICPGOLOG we have developed the language READYLOG and extended the ICPGOLOG interpreter implemented in ECLiPSe Prolog accordingly. READYLOG extends ICPGOLOG by possibilities for decision-theoretic planning as in DTGOLOG. This can be compared to the work of Soutchanski [37] as it deals with on-line planning and plan execution. Yet, our solution differs from Soutchanski's work by the direction from which the problem is approached.

Moreover, the concept of options has been integrated: defining local MDPs, the user can compute optimal policies for recognized sub-problems along with models about the effects when following such a policy in a certain state. The result can be used just like any other nondeterministic action and in particular the user can define options over options.

Motivated by the need to increase performance, a preprocessor has been implemented that translates READYLOG code to Prolog predicates. Finally, the preprocessor is also used to allow a user-friendlier syntax and to automate the computation of options from local MDPs.

We begin this chapter by presenting our approach of integrating decision-theoretic planning into an on-line interpreter. Thereafter, options and especially their construction within our interpreter are discussed. We finish the chapter with some details about the functioning of the preprocessor.

We use standard logical notation and present the definitions of predicates as formulas (the implementation in ECLiPSe Prolog can be found in the appendix). We assume standard semantics for arithmetics and lists. We use `typewriter`-font for implementational details and when presenting example programs.

5.1 Decision Theory

As discussed in Section 4.2.1, ICPGOLOG does not include nondeterminism. We reintroduced nondeterministic action selection and nondeterministic choice of argument with a semantics similar to that of DTGOLOG. Before we can formalize the MDP implicitly defined by a program, we have to decide how to model uncertainty in our system. Along with that comes the question of how to maintain full-observability. After we have presented our answers to these questions and have formalized the implicit MDP, we will direct our attention to questions arising from the new on-line context.

5.1.1 Modeling Uncertainty

Both, ICPGOLOG and DTGOLOG have a notion of uncertainty, though the way uncertainty is modeled differs with respect to the different applications. The two approaches can be described as follows.

- ICPGOLOG models uncertainty using $prob(p, \sigma_1, \sigma_2)$ statements, expressing that with probability p the sub-program σ_1, and with remaining probability $1 - p$ σ_2 is executed. Uncertainty of that kind is used to perform probabilistic projection which is done as follows: In parallel, using the construct $pconc$ for probabilistic concurrency, a program σ and a low-level model λ of the base actions of the system are interpreted. Thereby, the low-level model works as a simulator of what would actually happen. When the program executes an action, the model simulates its effects on the world, that is, it changes fluent values accordingly. Since the low-level model can be an arbitrary program, particularly able to use $prob$ also within $while$-loops, it is not possible to foretell the number of outcomes the simulation of one action may have. This causes problems for maintaining full-observability as we will see below.[1] Here are the advantages (+) and disadvantages (–) of this approach:

  + The effects of actions can be described by complex programs using all common programming constructs. This allows to model very complex effects.
  – The actions in the program and their effects described in the model are not linked. This makes it hard to decide when a sensing action should be executed to determine which outcome an action actually had.
  – The concurrent interpretation comes along with a rather huge computational overhead. Also, as the low-level model by itself is a program, it has to be *interpreted* to determine its impact on fluents. This is rather slow compared to other possibilities.
  – Care has to be taken by the user when implementing the low-level model to assure it is only executable when intended. It has to intervene in the program exactly when an action has to be simulated. This can be achieved using a certain architecture (see Section 4.3 of [18]).

- We described the way DTGOLOG models uncertainty in Section 4.2.2 leaving only the pros and cons to be mentioned:

  + It is straightforward to implement the possible outcomes/effects together with each nondeterministic action.
  + Full-observability is maintained.
  + The effects of the possible outcomes of an action are already represented by successor state axioms, which makes any more interpretation at time of projection unnecessary and thus saves time.
  – It is not possible to model complex outcomes. All effects have to be assembled into the successor state axioms of the outcome-describing primitive actions.

[1] Note that for the use of probabilistic projection as in ICPGOLOG, full observability is not an issue. This is, because the intention of the projection is not to construct a policy. See [22] for details.

We aimed at merging the advantages of both approaches. Our solution makes use of the preprocessor which we will present in detail in Section 5.3. The main idea is to generalize from nondeterministic primitive actions to nondeterministic procedures. At time of execution the normal body of the procedure is used, while for projection a model is applied. This leaves more freedom and includes the approach of DTGOLOG as a special case.

Stochastic Procedures

We first present the formal definition of stochastic procedures, before describing how the user can define them more abstractly.

A *stochastic procedure* is a procedure $p(\vec{x})$ defined by $proc(p(\vec{x}), b_p(\vec{x}))$, where $\vec{x}$ are the arguments of p, for which

- a precondition axiom $stoch\_proc\_poss(p(\vec{x}), s) \equiv \varphi_p(\vec{x}, s)$ exists, and
- an effect axiom of the form $stoch\_proc\_outcomes(p(\vec{x}), s, \Omega, \psi) \equiv \gamma_p(\vec{x}, s, \Omega, \psi)$ exists, where Ω is a list of *outcomes* and ψ is a sensing program which senses all necessary data to determine the outcome that actually happened. The list of outcomes has elements of the form (ω, pr, ϕ), where ω is an ICPGOLOG program describing the outcome, pr is the probability for this outcome, and ϕ is the sense condition for this outcome, that is, a condition that holds if and only if this outcome in the list has actually happened. The program ω has to be deterministic and in particular must not contain $prob$-statements.

This constitutes the counterpart to the $nondetActions(a, s, \Omega)$ predicate of DTGOLOG: we replace both, the action and the elements of the outcomes-list by procedures. Furthermore, we integrate the definition of sense conditions and a sense program into this definition, instead of keeping them seperate as in DTGOLOG (cf. Section 4.2.2). The semantics of these axioms is similar[2]: if procedure $p(\vec{x})$ is executed in situation s, the world will change to a new situation s', where s' is defined by $trans^*(\omega_i, s, \omega_{i_{final}}, s')$ with probability pr_i, where ω_i is the program of the i-th entry in list Ω, pr_i is the corresponding probability and $trans^*$ is the transitive closure of the $trans$-predicate (cf. Section 4.2.1). That is, the programs are "executed" by the trans predicate until a final configuration, where no more $trans$-steps are possible, is reached. In this configuration the remaining program $\omega_{i_{final}}$ is called final.

That way the outcome describing programs define a set of possible successor situations just like the outcome describing primitive actions in DTGOLOG. The difference is that entire procedures, as used here, allow more flexibility than single primitive actions.[3]

However, this is the *formal* definition of a stochastic procedure. We do not require the user to define these axioms, and in particular do not require him to implement any Prolog code. Instead, the user can do all definitions in a GOLOG like metalanguage. These will than be converted by the preprocessor to the above form, thereby also compiling as much as possible to Prolog to speed up the projection (see Section 5.3 for details about the preprocessor, its aim, and the way it works).

We now present a motivating example which is to present the intuition about the definitions. Afterwards we go into some technical detail of defining stochastic procedures.

[2]For a formal definition of the semantics of stochastic procedures see page 60.

[3]Without proving this claim any further, we remark, that this is because of the same reasons why while-loops are not first order representable.

Example 5.1.1 *For example, the following procedure model could be defined:*

```
1 proc_model( makeCareer,
2   [ if(educationLevel=high,
3        sprob([ (becomeProfessor, 0.4, employer=university),
4                (becomeManager, 0.6, employer=company)],
5              senseEmployer),
6        sprob([ (becomeFireman, 0.3, drive=fire_truck),
7                (becomePoliceman, 0.5, drive=motorcycle),
8                (becomeAmbulanceman, 0.2, drive=ambulance)],
9              senseVehicle) )] ).
```

The intuition is the following: the procedure `makeCareer` *has different outcomes depending on the level of education of the agent (we assume that* `educationLevel` *is a fluent and can have values* `high` *and* `low`*). If the agent is well educated, making career will make it a professor with probability* 0.4 *and a manager with probability* 0.6*. After performing this procedure in the real world, the agent can determine the actual outcome by sensing its employer: If it is a university, it must have become a professor. Otherwise, if it is a company, it turned out to be a manager. On the other hand, if the educational level is not high, the agent will either become a fire fighter, a policeman or an ambulance man.*

The following details are rather technical and can be seen as a reference for using the interpreter. They are in particular not essential for understanding the remainder of this or any following chapters.

The user can define any procedure to be stochastic by providing

- a precondition φ_p, defined by $proc\_poss(p, \varphi_p)$, and
- a *procedure model* m_p, defined by $proc\_model(p, m_p)$,

where φ_p is a legal ICPGOLOG condition stating when the procedure is executable.[4]

A *procedure model* m is a possibly empty sequence of

- primitive actions,
- tests $?(\varphi)$, where φ is a legal ICPGOLOG condition,
- conditionals $if(\varphi, a, b)$, where φ is a ICPGOLOG condition and a and b are procedure models, and
- an optional $sprob$-statement at the very end of the sequence.[5]

An $sprob$-statement[6] takes two arguments:

- a list of *outcomes* Ω where each element has the form (e, p, ϕ) with
  - e an arbitrary ICPGOLOG program describing the effects of this outcome,

[4] See [22] for a formal definition of an ICPGOLOG condition.

[5] As soon as an $sprob$-statement is hit, the sequence is cut after considering this statement. If in fact such a statement does not appear as the final element of the sequence, the preprocessor prints a warning to the screen that any later instructions are being ignored.

[6] Read s-prob-statement, for *sensible* prob-statement.

  - p the probability for this outcome,
  - ϕ a legal ICPGOLOG condition which holds if and only if this outcome in the given list has happened.

- a *sensing program* which is an arbitrary ICPGOLOG program which is understood to perform all required sensing actions to determine which outcome has happened.

The probabilities in list Ω are assumed to sum up to 1.0.[7] The $sprob$-statement has a similar semantics as the $prob$-statement of PGOLOG [20] and ICPGOLOG. The only differences are that in an $sprob$-statement we can have an arbitrary long list of alternatives, each carrying its own probability, and that full-observability is maintained by naming a sensing program (usually only one primitive action) and sense conditions for all outcomes.

From these definitions, the preprocessor creates a stochastic procedure, by generating the required axioms. In particular, all appearing conditions are compiled to Prolog code (see Section 5.3), which supersedes further on-line interpretation, saving time.[8]

While the previous example was particularly made up for illustration, to show the use of if-statements in such models, and to show a case where effectively more than one $sprob$-statement is permitted to appear, the following is an example that we actually applied in controlling robots.

Example 5.1.2 *We used the following procedure model at* ROBOCUP*2003 (Padua) Mid-Size tournament to describe our intuition about the possible outcomes when a robot tries to intercept the ball.*

```
1  proc_poss(intercept_ball(_Own, _Mode), not(f_ballInGoal)).
2
3  proc_model( intercept_ball(Own, _Mode),
4    [?(and([ AngleToBall = angle(agentPos, ballPos),
5             NewPose = interceptPose(AngleToBall),
6             NewPose = [X,Y,Angle]])),
7     sprob([
8             ([set_ecf_agentPos(Own, [X,Y], [0,0]),
9               set_ecf_agentAngle(Own, Angle, 0)],
10             0.2, isDribblable(Own)),
11            ([], 0.8, not(isDribblable(Own))) ],
12            exogf_Update)
13   ]).
```

The first definition expresses that intercepting the ball is possible if and only if the ball is not in one of the goals. The first part of the procedure model is a test in which the position is computed which would result from directly driving to the ball from the current position of the robot. In the consecutive `sprob`*-statement, two possible outcomes are listed: either the robot successfully intercepts the ball and ends up at the calculated position at the ball (*`set_ecf_agentPos(·)` *is a primitive action setting the agent's position to the given value) with a heading towards the ball (*`set_ecf_agentAngle(·)` *is a primitive action setting the agent's angle to the given value). Or the interception fails leaving the*

[7] Actually the preprocessor prints a warning to the screen if at compile time it can already recognize a violation here.

[8] Note that for these definitions there is no formal semantics defined, since they are converted to a stochastic procedure, for which we already described the semantics.

robot where it is. The rather low probability of 0.2 for a success expresses how difficult it is for the robot to securely intercept the ball. The condition `isDribblable(Own)` *holds if and only if the ball is close in front of the robot. The empty program in line 11, used to denote a failure, is probably not a good description of what really changes in the world if the interception fails. However, it is very complex to account for all ways the action could fail. Yet, for planning it mostly only matters that the robot remains without the ball.*

Also the user can define (optional) costs connected with a procedure. This is done by the predicate $proc\_costs(p, costs_p, \varphi)$ where φ is a condition under which the costs are $costs_p$.[9] For example, $costs_p$ can be a variable which depends on the condition φ. Then it is possible to connect different costs with different situations. For the example above, we assigned higher costs in situations where the ball is in front of the own goal, since intercepting the ball there is rather risky. Intercepting the ball in front of the opponents goal, on the other hand, was even assigned a reward (negative costs).

Explicit Events

In Section 3.1.2 we discussed two types of action models: implicit- and explicit event models. In a nutshell, implicit event models describe all events that can happen between two agent actions, as effects of the first action. Even if an event does not have any causal relation to that action. Explicit event models, on the other hand, describe such events as independent of agent actions by specifying their probability of occurrence and their possible effects depending on the system state. While the former is required for an MDP, the latter is more intuitive to implement. If certain assumptions are met, we described how an explicit event model can be transformed to an implicit one. In READYLOG we allow the user to define explicit events and making the required assumptions apply the described algorithm to automatically generate an implicit model from that[10]. Explicit events are in addition to stochastic procedures a way to express uncertainty of the system and can, for example, be used to model the behavior of other agents in a multi-agent system.

Explicit events are defined very similar to stochastic procedures, but they lack a procedure body, since such events are only intended for projection, not for execution. The predicates $event\_poss(\varepsilon, c_\varepsilon)$, $event\_model(\varepsilon, m_\varepsilon)$, and $event\_costs(\varepsilon, costs_\varepsilon)$ are used to define an event. Intuitively, the semantics is as follows [11]: at each step of the projection, for each defined event it is checked whether this event is possible in the current situation. If so, the projection is continued for each of its outcomes.

Then, for example, expected adversarial behavior can be modeled. Assume we observed that in ROBOCUP mid-size the opponent goalie always stays on one line with the ball. Then this can be modeled by something like the following:

```
1  event_poss( oppgoalie_event, true).
2  event_model( oppgoalie_event,
3               [ ?(and([position(opp_goalie) = [GX, GY],
4                        ballPos = [BX, BY]])),
5                 set_position(opp_goalie, [GX, BY])
6               ]).
```

[9] This will be converted by the preprocessor to a predicate $stoch\_proc\_costs(p, c, v) \equiv \dot{\varphi}$, where $\dot{\varphi}$ is logically equivalent to φ, but is implemented in Prolog (see Section 5.3).

[10] In fact, no explicitly represented implicit event model is created. Instead the necessary transformations are done on-the-fly in the projection algorithm.

[11] cf. page 57 for the formal semantics

The alternative would be to require the user to implement already an implicit event model, which in the given example would amount to folding this event into the models of all actions, as described earlier. Thus, adding a new event would always require modifications to all existing action models.

Explicit events can also be used to model some forms of concurrency and the occurrence of exogenous events.

The pros and cons of our approach to modeling uncertainty can be summarized as follows:

+ Full-observability is maintained.

+ Because every sub-program can be encapsulated into a procedure, for any sub-program a model can be defined. This enables the user to freely choose the level of abstraction at which he likes to model nondeterminism and, thus, on which level to perform planning.

+ Interpretational overhead is minimized.[12]

+ The definitions by the user are independent of Prolog. This is another step towards freely choosing in which programming language to implement the interpreter or even a compiler.

– The usage of constructs is restricted compared to general PGOLOG programs as can be used in the probabilistic projection of ICPGOLOG.

The restrictions of the last item arise from the need to maintain full-observability.[13]

Full-Observability

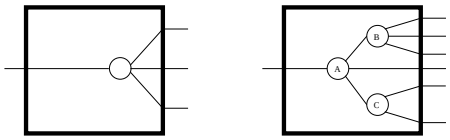
Some relevant considerations about full-observability, the ability of the agent to always accurately determine the current state, conclude the discussion of modeling uncertainty.

As in DTGOLOG, full-observability in READYLOG is provided by keeping track of what the agent does (confer Section 4.2.2). Recall, that this means that after performing an action which was declared nondeterministic, the agent performs one or more sensing actions to determine which of the possible outcomes has actually happened. The agent then always knows which primitive actions of the situation calculus have actually happened and thus knows the situation, that is, the state. Figure 5.1 (a) shows the case of uncertainty which we permit: the box symbolizes a nondeterministic action. Its procedure model, the content of the box, uses exactly one $sprob$-statement with three possible outcomes. After executing this action in the real world, the agent can use the sensing actions defined together with the $sprob$ to determine which outcome actually happened.

Suppose we permitted compositions of probabilistic branching, for example through sequences or nesting of $sprob$-statements, this is illustrated by Figure 5.1 (b). Then we would need to generate a sensing program and sense conditions to observe the overall outcome. However, the user only specified how the alternatives at one branching point (in one $sprob$-statement) can be distinguished. A naive attempt would be to sequence

[12] In fact, the Prolog code generated by the preprocessor is as fast as hand coded effects in DTGOLOG would be. See Section 5.3 for a quantitative comparison.

[13] cf. Section 4.2.2



(a) (b)

Figure 5.1: (a) a nondeterministic action with only one $sprob$-statement, indicated by the circle; (b) a nondeterministic action with nested $sprob$-statements

the sensing programs and conjoin the sense conditions of sequential branchings. Unfortunately, this does not work in general since an early sense condition might be voided by a later effect. Technically, this is true, because a nondeterministic action of any kind (treated as primitive or complex) is like a black box to the user and we need a sensing program and a sense condition that can be used afterwards to decide which outcome has happened, based only on the input. In particular, we cannot observe intermediate states within the box. In the figure this means that we have no possibility to sense the outcome of the $sprob$-statement A. This kind of uncertainty we do *not* permit. We only allow uncertainty like the one of Figure 5.1 (a), with at most one $sprob$-statement at the end of a sequence in a procedure model.[14]

Another approach to this problem which could be investigated in future work could be the following: Instead of focusing on the moment of branching, one could create conditions directly for the outcomes. This could for example be done by specifying a set of fluents and to determine the values for these fluents for each possible outcome according to the applied actions within the model (all described in one way or the other in the situation calculus). After executing the nondeterministic program part – action or procedure – the values for these fluents would have to be sensed and used to decide on the actual outcome. This approach is motivated by the question which of the listed outcomes describes the actual fluent values best. However, in realistic domains where fluents are commonly real valued, it is mostly unlikely that the values of the specified fluents in the model match with those in the real world. Thus, some kind of classification is required. This certainly goes beyond the scope of this work. In investigating this matter, it could be interesting to combine the research with on-line learning techniques to improve the given models.

5.1.2 Specifying the Implicit MDP

Now that we have settled how to model uncertainty and in particular nondeterministic actions, we can now in analogy to Section 4.2.2 specify the implicit MDP defined in a READYLOG program, albeit only briefly, focusing on the differences to DTGOLOG.

state space = situations : This point stays unchanged.

action space = stochastic procedures + primitive actions : In addition to the primitive actions, we have to formally add stochastic procedures to the set of MDP actions. This is because the models we have of them only apply to the procedures

[14]Note that in the earlier example on page 51, the two occuring $sprob$-statements appeared in disjoint cases of a conditional and in particular are neither nested nor in sequence.

as such, although the procedures usually contain primitive actions themselves. Hereby the later defined *options* are included as they are compiled to stochastic procedures by the preprocessor (see Sections 5.2, 5.3).

transition function = SSAs, procedure models, explicit events : The transition function is not as simple as in DTGOLOG, where the successor state axioms could directly be used for the transition function. For primitive actions it stays similar to the extent that the successor state axioms can be used, but now all possible explicit events have to be folded into the transition. It works as follows: after the effects of the primitive actions have been applied, a new situation is reached. The list of all events is traversed and all events that are possible in this situation are applied one after another, which leads to new situations and finally to the successor situations/states.[15]

This works similarly for stochastic procedures, though, for these no immediate successor state axioms exist. Instead the defined outcome programs have to be completely interpreted to obtain the possible successor situations. Due to the transition semantics of ICPGOLOG, this is, however, very easy as we can immediately use the transitive closure of the $Trans$ predicate. That is, $Trans$ is applied to each outcome program repeatedly, starting in the current situation, until a final configuration is reached. Afterwards the possible explicit events are processed, just like for primitive actions.

This rather intuitive description is formalized by the predicate $bestDoM$ in the next section.

reward function = reward - costs : In addition to fluents, ICPGOLOG offers to define *functions* that take a certain value depending on some condition. Formally a function is defined by $function(f, v, \varphi)$, with the semantics that the function f has the value v if the condition φ holds in the actual situation. In READYLOG, the user defines a special function $Reward$ which is understood to assign a value to each situation. For example $function(Reward, v, v = 10 - distance(Ball, Opponent\_goal))$ could be a reasonable reward function for the ROBOCUP Mid-Size League, based on the distance of the ball to the opponent goal.

Costs, as before, additionally depend on an action and can be defined for stochastic procedures. If no costs are defined for a procedure, it does not cause any costs.

5.1.3 Solving the Implicit MDP

The core extension we made to ICPGOLOG is precisely to solve MDPs implicitely defined in the program as described above and to execute the resulting policy. In this section we describe the developed and implemented algorithm for solving the MDP, in the next we show how to represent and execute the resulting policy.

Unlike the approach of Soutchanski [37], we do not use a semantics of continuous optimization. Recall, that his interpreter kept projecting the proximate future to create a policy for it, but then only performed the first step of it before restarting the policy construction. Our idea is closer to the construction of the incremental GOLOG interpreter in [12]. General program execution is defined by the transition semantics of ICPGOLOG. After each step of program interpretation, i.e., one call to $Trans$, the

[15] Note that by this semantics, the order of events matters.

action is executed in the real world. To interrupt the on-line execution in order to deliberate creating a plan, the user can use the new programming construct $solve$: a transition step of $solve(\delta, h)$ changes to off-line mode and starts the projection process for program δ up to a depth of h. Afterwards, the first step of the policy is executed, leaving the rest of its execution to the remaining program returned by $Trans$. Not all programming constructs of ICPGOLOG are supported within a $solve$-evaluation. We call a program that is allowed with $solve$-statements a *plan-skeleton*. We allow the following constructs in plan-skeletons: sequences, primitive actions, procedures (in particular stochastic procedures), if-then-else, tests, while-loops, waitFor, nondeterministic action choice, and nondeterministic choice of argument (with and without optimization, see below $pickBest$ and pi respectively). In particular, we do not allow constructs expressing concurrency, but allow some concurrency by defining explicit events. Below we discuss the most relevant constructs and explicit events in detail.

The projection process, optimizing over the proximate future, works as in DTGOLOG by applying decision tree search. This is implemented by the recursive predicate $bestDoM$. This predicate takes seven arguments, four of which we understand as input arguments, the remaining three we call output arguments. These are the arguments and their intuition:

1. The plan-skeleton δ to be optimized. It only makes sense to have plan-skeletons that mention nondeterministic choices, since it is exactly these we are aiming to optimize. That is, we want to decide which of the alternatives to take in order to achieve the greatest possible overall reward.
2. The situation s where the deliberation is started. At the root of the $bestDoM$-evaluation this is the situation where the execution of the global program was interrupted. Seen from the perspective of MDPs, this is the state.
3. The remaining horizon h until the optimization process stops to report the results.
4. The optimal policy π returned by the process.
5. The expected value v for following the above policy.
6. The probability p of successfully terminating the plan-skeleton.
7. Another input argument α indicating whether for this projection step the list of events has yet been processed or still has to be.

In the following, we will describe the definition of $bestDoM$ for the allowed programming constructs in plan-skeletons and for explicit events.

Explicit Events

The last argument of $bestDoM$ is a boolean: If it is $TRUE$, the interpreter initiates the processing of possible events. It starts by getting the complete list of defined events Υ, which is created at time of preprocessing:

$$\begin{aligned} &bestDoM(\delta, s, h, \pi, v, p, \alpha) \equiv \\ &\quad \alpha = TRUE \wedge h \geq 0 \wedge \exists(\Upsilon) events\_list(\Upsilon) \wedge \\ &\quad (\Upsilon \neq [\,] \wedge bestDoM\_event(\delta, \Upsilon, s, h, \pi, v, p) \\ &\quad \vee \Upsilon = [\,] \wedge bestDoM(\delta, s, h, \pi, v, p, FALSE)) \end{aligned}$$

If this list is empty, it simply proceeds with program δ. Otherwise, it calls an auxiliary predicate to process all entries of the list. Basically a tree as the one depicted in Figure 5.2 is spanned to create all of its leaves, where the normal projection continues. For

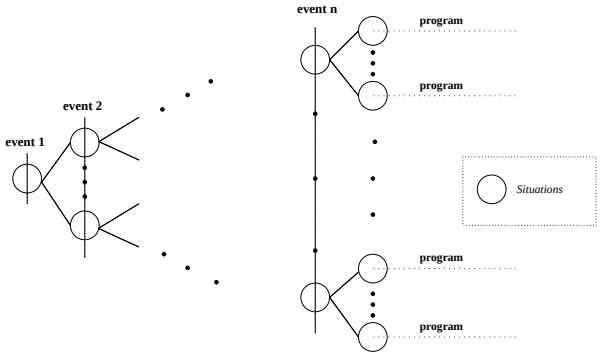


Figure 5.2: The tree created by the explicit events. At the leaves the ordinary optimization is continued.

spanning the tree we need two auxiliary predicates, one for iterating through the list of all events, the other for iterating through all possible outcomes of one particular event. The former is implemented as follows:

$$bestDoM\_event(\delta, [\,], s, h, \pi, v, p) \equiv$$
$$bestDoM(\delta, s, h, \pi, v, p, FALSE).$$

$$bestDoM\_event(\delta, [\varepsilon|\Upsilon], s, h, \pi, v, p) \equiv$$
$$(prolog\_event\_poss(\varepsilon, s) \wedge \exists(\Omega, \psi) event\_outcomes(\varepsilon, s, \Omega, \psi) \wedge$$
$$\exists(\pi_{list}, v_{list}) bestDoM\_event\_Aux(\delta, \Upsilon, \Omega, s, h, \pi_{list}, v_{list}, p) \wedge$$
$$\exists(r, c) prolog\_function(Reward, r, s) \wedge$$
$$(prolog\_event\_costs(\varepsilon, c, s) \vee \neg prolog\_event\_costs(\varepsilon, c, s) \wedge c = 0.0) \wedge$$
$$v = r - c + v_{list} \wedge \pi = [perform(\psi)|\pi_{list}]) \vee$$
$$(\neg prolog\_event\_poss(\varepsilon, s) \wedge bestDoM\_event(\delta, \Upsilon, s, h, \pi, v, p))$$

The first case, for an empty list of remaining events, constitutes a leaf of the tree. For each leaf, the program δ is further projected. In the second case, it is first checked if the event is possible in the actual situation. If so, the other auxiliary predicate is called to create all outcomes and go on with projection for each of it. Further, rewards and possible costs are computed and together with the subsequent value (v_{list}) combined to the returned value. To later recognize which event outcome has happened, similarly as for stochastic procedures, a sense program has to be executed at run time, which is indicated by the $perform(\psi)$ added to the policy.[16]

The second auxiliary predicate is defined as follows[17]:

$$bestDoM\_event\_Aux(\delta, \Upsilon, [(e, p_e, \phi)|\Omega], s, h, \pi, v, p) \equiv$$

[16] See Section 5.1.4 for details on the notion of a policy in READYLOG.

[17] For brevity we leave out the special case for the last outcome in the list. For the complete definition see Appendix A.3.

$$\Omega \neq [\,] \wedge \exists(\rho, s')trans^*(e, s, \rho, s') \wedge$$
$$\exists(\pi_{tree}, v_{tree}, p_{tree})bestDoM\_event\_Aux(\delta, \Upsilon, \Omega, s, h, \pi_{tree}, v_{tree}, p_{tree}) \wedge$$
$$\exists(\pi_{rest}, v_{rest}, p_{rest})bestDoM\_event(\delta, \Upsilon, s', h, \pi_{rest}, v_{rest}, p_{rest}) \wedge$$
$$v = p_e \cdot v_{rest} + v_{tree} \wedge \pi = [if(\phi, \pi_{rest}, \pi_{tree})] \wedge p = p_e \cdot p_{rest} + p_{tree}$$

The tuple (e, p_e, ϕ) describes an outcome, where e is a program describing the effects, p_e is the probability of occurrence and ϕ is a sense condition. These outcomes are generated by the preprocessor from the user's specifications. The transitive closure $trans^*$ is used to compute the resulting situation s' from applying that outcome program in the current situation. Thereafter, the remaining outcomes are processed in a recursive call and, starting in s', further projection is initiated. The last line contains the usual calculations for the returned expected value, policy and termination probability. Expected value and termination probability are simply the values of the subsequent projection (v_{rest}, p_{rest}) weighted with the probability for this outcome (p_e), plus the similarly calculated values for the other outcomes (v_{tree}, p_{tree}). The policy includes a conditional over the sense condition for this outcome to decide which sub-policy to take depending on the actual outcome at run-time.

Here the same concerns about full-observability apply as discussed above. We circumvent conflicting sense conditions of outcomes of different events, by requiring independent events. This is under the responsibility of the user.

Nondeterministic Choice

By using the construct $nondet(\Delta)$, where Δ is a list of plan-skeletons, the user can leave a decision to the agent, who is forced to decide depending on the expected reward for each choice.

$$bestDoM([nondet([\delta])|\delta'], s, h, \pi, v, p, \alpha) \equiv$$
$$\alpha = FALSE \wedge h \geq 0 \wedge \exists(\pi_{rest})bestDoM([\delta|\delta'], s, h, \pi_{rest}, v, p, FALSE) \wedge$$
$$\pi = [match(nondet(0))|\pi_{rest}]$$

$$bestDoM([nondet([\delta|\Delta])|\delta'], s, h, \pi, v, p, \alpha) \equiv$$
$$\alpha = FALSE \wedge h \geq 0 \wedge \Delta \neq [\,] \wedge$$
$$\exists(\pi_1, v_1, p_1)bestDoM([\delta|\delta'], s, h, \pi_1, v_1, p_1, FALSE) \wedge$$
$$\exists(\pi_2, v_2, p_2)bestDoM([nondet(\Delta)|\delta'], s, h, \pi_2, v_2, p_2, FALSE) \wedge$$
$$(greatereq(v_1, p_1, v_2, p_2) \wedge \pi = [match(nondet(0))|\pi_1] \wedge p = p_1 \wedge v = v_1 \vee$$
$$\neg greatereq(v_1, p_1, v_2, p_2) \wedge \exists(n, \pi_{2_{rest}}, n')\pi_2 = [match(nondet(n))|\pi_{2_{rest}}] \wedge$$
$$n' = n + 1 \wedge \pi = [match(nondet(n'))|\pi_{2_{rest}}] \wedge p = p_2 \wedge v = v_2)$$

The first case, for the last entry in the list, is rather trivial. The only thing to note is that in further projection events keep being ignored. We understand events to happen, if actually possible, only once at the beginning of each situation. Thus, we change the value for the argument indicating whether events should be preocessed or not only when reaching a new situation, after a primitive action or a stochastic procedure.

The second definition takes care of the branching: for the given alternative the further projection is started and the predicate is recursively called for all remaining choices. Based on the returned values, it is then decided which of the alternatives to *recommend*, that is, to write to the policy. This is done by the predicate

$greatereq(v_1, p_1, v_2, p_2)$, which expresses that the first expected value/termination probability-combination is preferred to the second. Basically the two expected values are compared, as long as the corresponding termination probabilities are greater zero. In the policy the index of the selected choice in the list is annotated.[18]

Nondeterministic Parameter Choice

Imagine a robot is to perform a certain action, but at least for one of the arguments it is unclear which would be the best value to take from a certain set of possible values. This could be modeled by using a $nondet(\Delta)$-statement where the list Δ contains the action for each of the possible arguments. However, especially for large sets of possible values and for more complex sub-plan-skeletons than just single actions, it seems convenient to offer a more compact way for describing this problem. The construct $pickBest(f, r, \delta)$ decides which replacement for all occurrences of f in plan-skeleton δ is the best if the replacement is taken from the set r. In our implementation, r is defined by a list of arbitrary elements and for two integer numbers `a` and `b` the expression `a..b` can be used to denote the range of integer numbers from `a` to `b`. The $bestDoM$ for $pickBest(f, r, \delta)$ works by simply substituting all possible values for all occurrences of f in δ, performing projection for all of these, and afterwards selecting the best one. Since this is very similar to the definition for the $nondet(\Delta)$-statement, we omit the details for this expression (see Appendix A.3 for details).

Stochastic Procedures

Stochastic procedures are treated by $bestDoM$ very similar to events. In fact, the only difference is that stochastic procedures are called from the plan-skeleton, while events just happen occasionally when they are possible. When they are not, it just means that nothing happens. However, if a called stochastic procedure is not possible, the policy has a dead end and projection terminates – for this branch.

$$
\begin{aligned}
&bestDoM([a|\delta], s, h, \pi, v, p, \alpha) \equiv \\
&\quad h > 0 \wedge stoch\_proc(a) \wedge \\
&\quad (stoch\_proc\_poss(a[s], s) \wedge \exists(\Omega, \psi) stoch\_proc\_outcomes(a[s], s, \Omega, \psi) \wedge \\
&\quad \exists(h') h' = h - 1 \wedge \\
&\quad \exists(\pi_{rest}, v_{rest}) bestDoM\_stoch\_Aux(\Omega, \delta, s, h', \pi_{rest}, v_{rest}, p) \wedge \\
&\quad \exists(r, c) prolog\_function(Reward, r, s) \wedge \\
&\quad (stoch\_proc\_costs(a[s], c, s) \vee \neg stoch\_proc\_costs(a[s], c, s) \wedge c = 0.0) \wedge \\
&\quad v = r - c + v_{rest} \wedge \pi = [match(stoch\_proc(a[s])), perform(\psi) | \pi_{rest}] \vee \\
&\quad \neg stoch\_proc\_poss(a[s], s) \wedge prolog\_function(Reward, v, s) \wedge \pi = [\,] \wedge \\
&\quad p = 0.0)
\end{aligned}
$$

Where $a[s]$ is the procedure call with all arguments evaluated in situation s according to the situation calculus. The second part of the disjunction shows the case when the stochastic procedure is not possible: The reward for the current situation still is earned, but the policy from here is empty and the probability of successful termination is zero.

[18] Readers familiar with the policy representation of DTGOLOG, which is simply a legal DTGOLOG program, might be wondering about the policy notation used. They are referred to the next section, where we argue for a new kind of policy representation and policy execution.

In the other case, again an auxiliary predicate is used, which, once more, we only show partially:

$$
\begin{aligned}
&bestDoM\_stoch\_Aux([(\omega, p_\omega, \phi)|\Omega], \delta, s, h, \pi, v, p) \equiv \\
&\quad h \geq 0 \wedge \Omega \neq [\,] \wedge \\
&\quad (\exists(\rho, s')trans^*(\omega, s, \rho, s') \wedge \\
&\quad \exists(\pi_{rest}, v_{rest}, p_{rest})bestDoM(\delta, s', h, \pi_{rest}, v_{rest}, p_{rest}, TRUE) \wedge \\
&\quad \exists(\pi_{tree}, v_{tree}, p_{tree})bestDoM\_stoch\_Aux(\Omega, \delta, s, h, \pi_{tree}, v_{tree}, p_{tree}) \wedge \\
&\quad \pi = [if(\phi, \pi_{rest}, \pi_{tree})] \wedge v = v_{tree} + p_\omega \cdot v_{rest} \wedge p = p_{tree} + p_\omega \cdot p_{rest} \vee \\
&\quad \nexists(\rho, s')trans^*(\omega, s, \rho, s') \wedge bestDoM\_stoch\_Aux(\Omega, \delta, s, h, \pi, v, p))
\end{aligned}
$$

What $bestDoM$ does here, after all, is to start the projection for each possible outcome and compute the *expected value* and *expected termination probability* over them according to their probabilities of occurrence. Furthermore, using the sense conditions, the policy is extended by a conditional branching to the corresponding sub-policy based on the sensing result.

Figure 5.3 shows an example of the kind of tree created by interleaving nondeterministic (user) choices and nature's choices. A policy could be depicted similarly, but reducing the outgoing edges from the boxes (user choices) to exactly one. The optimization algorithm computes the maximum over all sub-trees in the boxes, while in the circles it calculates the expected values over the sub-trees. The tree is traversed depth-first, leaving a branch when either the horizon is reached or a state is entered where no more actions are possible. Other ways of traversal could be imagined and investigated in future work, for example to create an anytime algorithm. We come back to this point in Section 7.

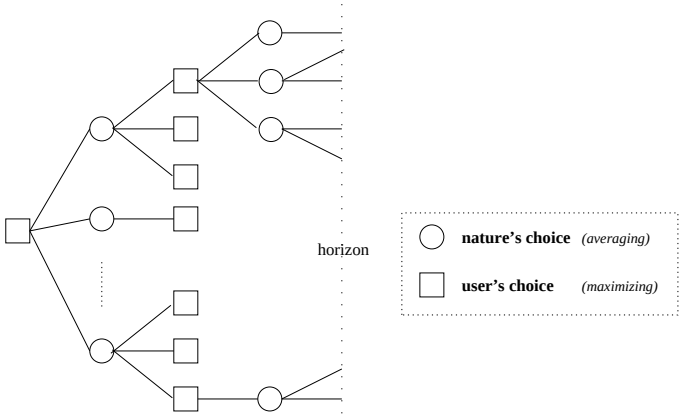


Figure 5.3: An example tree created by nondeterministic choices (boxes) and stochastic procedures (circles). Roughly, in the circles the expected value of all children is computed and in the boxes the maximum is chosen.

Other Supported Constructs

The yet presented constructs characterize the planning ingredient to our language. The programming ingredient is characterized by the remaining supported constructs, which are the following:

Sequence As in ICPGOLOG notated as a list `[a,b,..]`.

$?(\varphi)$ If the condition φ holds, the projection proceeds. Otherwise the branch is cut, returning an empty policy, zero termination probability and only the local reward.

$if(\varphi, \delta_1, \delta_2)$ If condition φ holds, the projection proceeds with sub-plan-skeleton δ_1, else with δ_2. In the policy, the truth value of the condition is annotated together with the remaining policy. See Section 5.1.4 for the reasons for that. As in ICPGOLOG we use $if(\varphi, \delta)$ as an abbreviation for $if(\varphi, \delta, [\,])$.

$while(\varphi, \delta)$ While condition φ holds, the plan-skeleton δ is projected over and over again. In the policy, each time the loop restarts is annotated. As in ICPGOLOG, we use $loop(\delta)$ as a shorthand for $while(TRUE, \delta)$.

$waitFor(\varphi)$ This advances the time, i.e. the value of fluent $start$, to the least time point where the temporal condition φ holds. If no such time point exists, the branch is cut. As in ICPGOLOG, we use $whenever(\varphi, \delta)$ as an abbreviation for $while(TRUE, [waitFor(\varphi), \delta])$ which performs δ whenever φ holds.

$pi(x, \delta)$ Technically, this sets all occurrences of x in δ to a fresh variable. From the logical point of view this statement equals $\exists(x)\delta$.

Primitive actions If the primitive action is possible, the projection continues for the successor situation $do(a, s)$, where a is the primitive action and s is the current situation, and adds the reward for the current situation. If it is not possible, the branch is cut. Primitive actions and stochastic procedures are, further, the only constructs which decrease the remaining horizon.

Procedures are supported as usual in GOLOG by macro expansion, that is, the procedure name is syntactically replaced by its body. One change, however, taken over from ICPGOLOG is that the actual parameters of a procedure call are already evaluated at the moment of replacement.[19]

Remember that these are only the constructs allowed in a plan-skeleton, used in a $solve$-statement. In particular, we do not reduce the set of allowed constructs in general, that is, every legal ICPGOLOG program is also a legal program in READYLOG.

Concurrency

Different from ICPGOLOG programs, in projection we do not allow constructs using concurrency, which are $pconc(\delta_1, \delta_2)$ and the macros $tryAll$ and $withPol$. This is due to similar reasons why Grosskreutz [18] removed nondeterminism in conjunction with concurrency and continuous change. The problem is to find a reasonable semantics for nondeterminism in concurrency. Concurrency in the model about the world is, however, supported by means of explicit events.

[19] In other GOLOG versions like in the original, actual parameters were only substituted for the occurrences of the formal parameters in the procedure body.

Optimize

In addition to the $solve(\delta, h)$-statement, READYLOG offers the statement $optimize(\delta, h, h_{exec})$ which equivalently starts the projection algorithm and performs the first step of the resulting policy. The difference comes with the semantics of policy execution: After a number of h_{exec} steps have been performed from the policy, the remaining part is dropped and instead the remaining program is newly optimized. Then, for example, $optimize(\delta, h, 1)$ would find a policy for plan-skeleton δ for h steps into the future, but then only execute the first of these steps and then restart finding a policy for what remains, again up to the given horizon. This special case, where h_{exec} equals one, is exactly the mode of operation Soutchanski [37] applies as the general case in his approach.

5.1.4 Policy

A policy describes the "best" way of behaving in order to achieve a high reward. A policy can be considered the *solution* of an MDP and needs to be represented and brought to execution, which we will turn to now.

In DTGOLOG, a policy is simply a GOLOG program. The intention is, that this program is computed as described and then executed in the real world *instead* of the original program, which may have contained nondeterminism. Unfortunately, the models of the world that are used for planning tend to be imperfect. That is, sometimes the world simply evolves different from what was considered possible. As discussed with the qualification problem, it is generally impossible to guarantee that a policy achieves its aim. However, since DTGOLOG is an off-line interpreter, the possibility of replanning in case something goes wrong, simply does not exist.

In our case, though, we are free to interleave plan generation and plan execution, and it *is* thus possible to react upon unexpected evolution. To do so, we cannot, however, represent a policy simply by a program anymore. Moreover it is necessary to somehow annotate the assumptions that have lead to this policy.

Example 5.1.3 *Consider the problem of running to the ball in the* ROBOCUP *Simulation League. The following program would usually be a promising approach to this problem:*

```
while(not(atBall), if(angleToBall > 20, turnToBall, dash))
```

With this deterministic program, the agent repeatedly dashes towards the ball if the angle to the ball is not greater 20 degrees. Otherwise it turns to the ball. Imagine this program is used within a plan-skeleton. Then the policy returned would be based solely on the applied model of the world. In fact, the policy that would be returned by DTGOLOG *for such a program would be some sequence of* `turnToBall`*s and* `dash`*es. However, this particular sequence would in the simulation league almost never get the agent to the ball, as the uncertainty in this domain is too complex to be modeled entirely in* DTGOLOG*. Thus, the policy would perform unnecessarily bad compared to the original program. The reason is mainly because the policy* commits *to the model and is unstable against changes. In particular, the involved conditions (*`atBall` *and* `angleToBall > 20`*) are evaluated based on the model and the resulting recommendation is not conditioned on the result.*

We propose a completely different representation for policies. The aim in developing this, was to increase stability and flexibility upon changes. Instead of creating a policy that itself is executable, we understand a policy only as an advisor for a particular program. The concept is best described by a metaphor: Consider a rally driving team, that is, a pilot and a his co-pilot. The pilot will be the program executor and the co-pilot will be the planner and provide the policy. The co-pilot has a map which symbolizes the model of the world. Based on the map he will continuously search the way ahead to determine how to steer at crossroads and other alternative ways. Yet, he will not steer for himself, but instead only tell the pilot how to decide when choices arise. Now, if something goes wrong, for example a road shown in the map is not usable due to weather conditions, the team has to reconsider its plan. While in this very example it is obvious that the original plan has to be discarded, because one of its elements is not possible, in general it might not always be that clear. Imagine the plan is to repeatedly go far left when choices arise, but different from the map a new junction to the left exists somewhere. Then applying the plan is still possible, but the rally team would go the wrong way without noticing. Thus, the task is to decide whether a plan is still valid or should be reconsidered.

We suggest the following: we keep executing the original program and only query the policy for advice when nondeterministic choices arise. To do that, we need to *synchronize* the policy with the program, in order to find out into which branch of the policy we have to go after some stochastic event, and to notice when the policy gets illegal. It is, thus, not enough to write a policy simply as the sequence of decisions to be made at the nondeterministic choices. Instead, all other elements should also be annotated. For some of the constructs, we already mentioned their annotation in the policy. In the following we give a complete list. Three different types of entries in the policy exist: $match(\cdot)$ is used to synchronize program and policy and for giving advice, $perform(\cdot)$ is used to indicate that the policy needs to perform a sensing program in order to know which way to go, and $if(\cdot)$ is used to branch in the policy depending on the sensing outcome.

- $nondet(\Delta) \rightarrow match(nondet(i))$: where i is the index of the best alternative from the list Δ.
- $pickBest(f, r, \delta) \rightarrow match(pickBest(v))$: where v is the best value to pick for f from the set r.
- stochastic procedure $a \rightarrow$ $match(stoch\_proc(a)), perform(\psi), if(\phi_1, \pi_1, if(\phi_2, \pi_2, ..))$: where a is the name of the stochastic procedure, ψ is the corresponding sensing program, ϕ_i is the sense condition for the i-th possible outcome, and π_i the corresponding sub-policy. If in reality the procedure is not possible, the policy is recognized to be illegal.
- explicit event $\rightarrow perform(\psi), if(\phi_1, \pi_1, if(\phi_2, \pi_2, ..))$: As for stochastic procedures, we need to branch in the policy depending on the actual outcome of an event that happened.
- $?(\varphi) \rightarrow match(?)$: The policy is invalid if the condition φ does not hold in reality.
- $if(\varphi, \delta_1, \delta_2) \rightarrow match(ifCond(\bar{\varphi}, \pi))$: where $\bar{\varphi}$ is the truth value of condition φ in the model, and π is the corresponding sub-policy. If in reality the truth of the condition changes, the policy is voided.

- $while(\varphi, \delta) \rightarrow match(while(\pi))$: If the condition holds, this entry is made to the policy. Otherwise, there is no entry for this construct. That is, the policy for this construct ends (is *final*) and can go on for a possible follow-up construct if used in a sequence.
- $waitFor(\varphi) \rightarrow match(waitFor(\varphi))$: If a least time point for the condition exists, this entry is added. Otherwise, no entry is made.
- primitive action $a \rightarrow match(prim\_action(a))$: If the action in the real world is not possible, the policy is discarded.

Whenever the policy gets illegal, re-planning is initiated for what remains.

This complex treatment is especially necessary, as in READYLOG like in ICP-GOLOG we allow for exogenous actions. As said before, these can update the agents view of the world without being triggered by the agent himself. That makes them different from sensing actions. In Soutchanski's approach [37], this matter is not treated, simply because he does not allow for exogenous actions. The only way the knowledge changes in his interpreter, is by performing sensing actions, which are called by the agent. Soutchanski suggests to start planning only right after sensing has been done and up to the next sensing action. However, this is unrealistic for dynamic domains: to perform well, frequent updates of the world model are necessary. In ROBOCUP it is commonly about once every 100 milliseconds. Yet, the described approach would restrain planning to only 100ms into the future, which does not make sense. In fact, that would only result in almost purely reactive behavior. After all, we are aiming at the contrary, namely deliberation. This clarifies why the approach described in [37] does not apply for our requirements.

Policy Execution

The execution of a policy amounts to executing the program, synchronously advancing the policy, and querying the policy for advice upon nondeterministic choices. The execution is defined in the transition semantics using the new construct $applyPolicy(\delta, \pi, h_{exec}, h_{replan}, h_{reexec})$:

- δ is the program to execute. This is exactly the plan-skeleton that was either used with $solve$ or $optimize$.
- π is the policy in the described notation.
- h_{exec} is the remaining horizon of execution.
- If the execution horizon is reached ($= 0$), re-planning is initiated for the remaining program with h_{replan} as planning horizon.
- The newly generated policy is then again executed with the new horizon of execution h_{reexec}.

The difference between $solve(\cdot)$ and $optimize(\cdot)$ comes from the values initially set for these parameters: $solve(\cdot)$ sets h_{exec} and h_{reexec} to -1. Thereby the policy is executed until it terminates naturally, either successfully or unexpectedly in which case re-planning is done. After a call to $optimize(\cdot)$, these values get set according to the parameters of the call.

The execution works as follows. In the general case, a $Trans$ step of the program is made and matched against the entry in the policy. If policy and program do not match, the policy is cut and re-planning is started. However, there are some special cases to be treated:

- When executing a *stochastic procedure*, the real procedure body is used, while in projection only the model for this procedure was considered. Thus, there is only one entry in the policy for matching the procedure call ($stoch\_proc(a)$) followed by the sensing and branching to be done after the procedure. Hence, the steps taken in the body of the procedure cannot be matched. To indicate that, when expanding the procedure body, it is encapsulated into an $ignore(\cdot)$-statement which is written to the program. Then $applyPolicy$ simply performs the program inside without trying to match.
- After a stochastic procedure has been executed completely, the sensing program within a $perform(\cdot)$-statement is executed.
- Following the sensing, the branching upon the outcome is done in the policy by interpreting an $if(\phi_i, \pi_i, \pi')$-statement. If the sense condition ϕ_i holds, the remaining policy is set to π_i, otherwise it is set to π' (which again can be an if-statement).
- When the program hits a $nondet(\Delta)$ decision, the policy, if nothing went wrong, is at a $match(nondet(i))$ entry. Simply, then, the element of list Δ with index i is chosen.
- Similarly a $pickBest(f, r, \delta)$ is resolved through a $match(pickBest(v))$ entry, where v is the value to chose from r.

5.2 Options

In Section 3.2 we introduced the concept of options and showed their potential for speeding-up MDP solutions. The idea was to abstract from the level of primitive actions, understood as actions that the agent is able to perform immediately, to more complex actions. These complex actions were called options or macro-actions.

We now want to present how we enable the definition and use of options in READYLOG. The aim, as in the literature, is to speed up the solution of the (implicitely) defined MDP. We will show, how options nicely extend the set of nondeterministic actions presented earlier in this chapter.

The crucial point for using options in planning, was the need of having models for them describing their impact on the world. In fact, what we defined as stochastic procedures, fits this notion: stochastic procedures are complex procedures defined over primitive actions, and a model for them is provided by the user. However, we do *not* understand stochastic procedures as options. Instead, we want to reserve the term *options* for macros for which the corresponding model has been created automatically based on the contained constructs and actions. These actions can themselves be options or stochastic procedures, allowing hierarchical option construction.

The hierarchy is defined in levels of actions:

Definition 5.2.1 *A* basic action *is either a primitive action or a stochastic procedure.*

With this in place, we can define options in our context:

Definition 5.2.2 *An* option of level n *is a tuple* $\langle \varphi, \pi \rangle$ *where* φ *is a condition describing the situations where the option is possible, called the* initiation situations*, and* π *is an* ICPGOLOG *program that is executable in all initiation situations and only contains basic actions and options of levels* $\leq n$*.*

This definition requires that the policy-program π for every situation where the option is possible names an action of lower level which to take in that situation. This could for example be a decision tree of the form $if(\phi_1, a_1, if(\phi_2, a_2, ..))$ where the ϕ_i describe situations and where for each initiation situation at least one of these conditions holds.

READYLOG supports the automatic creation of options via the solution of local MDPs. We take over this idea from Hauskrecht et al. [21] as presented in Section 3.2.2: The idea was to define local MDPs that describe sub-problems of the global task. Their solutions define options.

5.2.1 Defining local MDPs

The automatic creation of options in READYLOG is restricted to local MDPs with finite state space. This is in contrast to the global optimization mechanism, which can work on infinite state spaces as it exploits the knowledge about the starting point. However, options are not meant to start always from the same point. Therefore we cannot exploit such information. Furthermore, explicitly enumerating the state space avoids the problem of visiting apparently equal states several times, as discussed in Section 4.2.2. The definitions to be made by the user, which we will present in the following, carry the prefix $option\_$. Still, these only define the local MDP from which an option later can be created. They do not define the option itself.

The most central idea in defining local MDPs within our context, is to create a state space which is linked to the global task by explicit *mappings*. Three mappings are required:[20]

1. *situations → states*
   For each situation where the option shall be applicable, the corresponding state of the local MDP has to be known. The representation for states can be arbitrary, but it is suggestive to use a factored representation based on variables. In the example of Figure 4.1 on page 43, which we used to illustrate the difference between intuitive states and situations, a reasonable state description would be the position of the robot and the dirtiness of the rooms. That could be written simply as a list of tuples $[(Pos, p), (DirtLeft, l), (DirtRight, r)]$ where p, l and r are the values of these state variables. Then the mapping would be straightforward, by only evaluating the state variables in the actual situation, and notating the values in this representation.

2. *states → situations*
   Since there are usually infinitely many situations that would be mapped to the same state, we naturally cannot have a surjective mapping from states to situations. Fortunately this is not necessary, if the states depend on some crucial state variables as suggested above. Then from the values of these variable, a

[20] We below alleviate this complicated matter for the user by offering help in defining these mappings when certain conditions are met. However, first, the general case is discussed.

situation can be created which simply has these values set for the corresponding fluents. For the example above, we could, for instance, create the situation $do(setPos(p), do(setDirtLeft(l), do(setDirtRight(r), S_0)))$, where the primitive actions simply set the corresponding functional fluent to the given value.

3. *states → conditions*
   Further, a mapping from states to conditions is required, which forms the basis for full-observability in the local MDP. It is needed for option construction and is essentially the inverse direction from the first mapping. As it is more intuitive to explain this matter when needed, we skip the details for now.

The user defines all three mappings by one statement:

$$option\_mapping(o, \sigma, \Gamma, \varphi)$$

Each situation in which φ holds corresponds to state σ. Γ is a list of primitive actions that, when applied in an arbitrary situation (e.g. S_0), leads to a situation where the condition holds and thus corresponds to the given state. One example would be the following, which recognizes a fluent for the two-dimensional position (`pos`) as the only discriminative state variable:[21]

```
option_mapping( option1, [(pos, [X, Y])],
                [set(pos, [X,Y])], pos = [X,Y]).
```

The primitive action `set(FluentName, Value)` simply sets the value of the fluent to the given value. This action, which requires some special treatment in the interpreter, can be used with any fluent.

In the example above, the fluent pos is the only one that matters. In cases like this, where one or more fluents can be used for a *factored representation* [6] of state space, the mapping is very straighforward. To alleviate the task of designing it in those cases, we offer the user to, instead of defining the above mapping by hand, define a predicate $option\_variables(o, \Sigma)$, where o is the name of the option and Σ is a list of fluents. These fluents are understood to be state variables, such that two states are equal if and only if the values of these variables coincide. From that predicate, the preprocessor can create the required mappings, following the pattern of the above example.

This seems suggestive in discrete domains. However, if fluents take continuous values, this way of creating a state representation would not lead to finite regions of state space as we require it.[22] Then, the more general way for defining these mappings can be used by the user to realize some kind of abstraction he might have in mind. This issue relates to future work, where the applicability of options shall be extended to continuous domains (see Section 7).

The remaining steps in defining a local MDP are rather intuitive. As Hauskrecht et al. [21] we define regions of state space where an option is applicable:

$$option\_init(o, \varphi)$$

where the condition φ holds in only those situations where option o is applicable. This condition, however, has to be carefully designed. It must hold if and *only if* the option

[21] This is presented in the (Prolog) notation of our implementation.

[22] Although discrete domains may also be infinite, we only require that regions of state space are finite. Intervals of discrete domains are finite, but intervals of continuous (real valued) domains are not.

is applicable. We require this to be able to construct a set of situations that, when mapped to states, covers the entire region of states where the option is applicable. Let us explain that by an example:

```
option_init( option1,
             and([ pos = [X, Y], domain(X, [0..5]),
                   domain(Y, [0..10]), inRoom(1) ]) ).
```

Here `option1` is only applicable if the position fluent equals the list `[X, Y]` and `X` is an integer number from the interval `[0, 5]`, `Y` is an integer number from the interval `[0,10]` and the position is inside room 1. What makes this condition special are the `domain(·)` terms. They are special to our implementation and bind a variable to a finite domain using the finite domain library of ECLiPSe. It is important to note that without, the condition would not meet the requirement from above, since also infinitely many other values for `X, Y` would then make the condition true (e.g. `X = 2.71828, Y = 3.14159`). The `domain(V, R)` statement here proves useful, as it may bind a variable `V` to any value in the range `R`, where this range can be an arbitrary mixture of integer intervals (`a..b`) and any other terms.[23]

We call the set of states defined by $option\_init(\cdot)$ *initiation states* and all corresponding situations *initiation situations*.

To assure full-observability in the local MDP, the agent may require sensing to find out which state it is in when executing the resulting policy/option in the real world. The corresponding sensing program for an option is defined by

$$option\_sense(o, \psi)$$

An option is only executable in the initiation states, which implies that outside of this set an option immediately terminates. This defines the exit periphery like in Section 3.2.2 [21]: A non-initiation state that can be reached from an initiation state we call *exit state*. To the exit states the user can assign values, which we want to call *pseudo-rewards*. They are used only for solving the local MDP and are intended for creating goal-directed behavior.

$$option\_beta(o, \varphi, v)$$

states that if condition φ holds in an exit state it gets v assigned as a pseudo-reward. Otherwise, it gets a pseudo-reward of zero. In our implementation several such $option\_beta(\cdot)$ definitions may exist for different states or sets of states.

The set of actions from which the agent may choose at each stage of a local MDP of level n are defined via an option-skeleton: An *option-skeleton of level* n is either

- an action of level less than n,
- a nondeterministic choice $nondet(\Theta)$ where the elements of list Θ are again option-skeletons of level $\leq n$, or
- a conditional $if(\varphi, \theta_1, \theta_2)$ where φ is a condition and θ_1, θ_2 are option-skeletons of level $\leq n$.

In particular, no sequences are allowed. This restriction is required to maintain the Markov property. With sequences, the available actions in a stage would not only depend on the current state, but also on the history of actions. Compared to ordinary

[23] cf. $pickBest(f, r, \delta)$ in Section 5.1.3.

MDPs, an option-skeleton again provides means of incorporating domain knowledge to reduce computational complexity by offering conditionals. The specification of an option-skeleton for a local MDP is done by the predicate

$$option(o, \theta, \gamma)$$

where $0.0 < \gamma < 1.0$ is a discounting-factor which is used in the solution algorithm described in the next Section.

We are now ready to formalize local MDPs of level n:

Definition 5.2.3 *A* local MDP O of level n *is a tuple* $\langle S, A, T, \mathcal{R} \rangle$ *with the components defined by*

$S =$ *initiation states* $\cup$ *exit states* $\cup \{\alpha\}$*;*

A*: the actions mentioned in the option-skeleton, all of level less than n;*

Tr*: the outcomes for the used stochastic procedures and the successor state axioms of used primitive actions, both folded with possible explicit events (cf. Sections 5.1.2, 3.1.2); further, state* α *is absorbing and any action taken in an exit state leads to* α*;*

$\mathcal{R}$*: local pseudo-rewards + global rewards – global costs; for state* α *this value is zero.*

Again there is no explicit representation for the transition function and in particular the folding of events is not performed explicitly, but implicitly in the algorithm.

We use value iteration to solve such an MDP. However, we want to use the solution for solving a global task. Thus, it does not suffice to solve the MDP. We require a model of the possible effects when following the acquired solution. Value iteration for the infinite horizon problem already provides the expected value for taking the option in any of the initiation states. Still missing are the probabilities of ending up in a certain exit state depending on the state where the option is taken. Since we are not limiting the initiation of an option to some kind of entrance states as in [21], we cannot reduce the computation of effects to these. In this way, our approach is a mixture of the ideas of Sutton et al. [38] and those of Hauskrecht and his colleagues.

5.2.2 Local MDPs: Solution and Model

We have implemented a slightly modified value iteration algorithm plus an iterative algorithm to compute the probabilities of ending up in a certain exit state of a local MDP depending on where the (optimal) policy is taken.

The modification of value iteration is in order to take both, pseudo-rewards and globally defined rewards into account. Also the implicit folding of events into actions was added following the algorithm described in Section 3.1.2 [6]. The algorithm runs iteratively. For each iteration h, a state s possesses a tuple $\langle\, r_h(s), v_h(s), \pi_h(s), \Pi_h(s) \,\rangle$, where

- $r_h(s)$ is the expected reward when taking the option in this state,
- $v_h(s)$ is similarly the expected pseudo-reward,
- $\pi_h(s)$ the policy, that is, the action to take in state s,

- $\Pi_h(s)$ the *probability list*, listing all possible successor states of s together with their probabilities.

Initially for all initiation states $r_h(s)$ is set to the global reward defined for this state, $v_h(s)$ is zero, the policy is nil, and the probability list is empty. For exit states s' $r_h(s')$ is set to zero, $v_h(s')$ is set to the pseudo-reward as defined by $option\_beta(\cdot)$, the policy is nil, and the probability list empty.

The iteration operates on the set of initiation situations constructed from the condition given by the predicate $option\_init(\cdot)$. For each such situation the corresponding (initiation) state can be created by using the defined mapping from situations to states. The iteration step h for situation s is defined through a new predicate $bestDoMOpt$ with ten arguments:

1. the name of the option, o,
2. an option-skeleton θ,
3. a boolean α indicating whether events have yet been processed or not,
4. a situation s from which the state is determined,
5. the iteration number h,
6. the expected global reward r for this state,
7. the probability list Π for this state,
8. the depth d, which is the number of steps taken into the option-skeleton (with regard to nestings),
9. a policy π, which is an action of level $< n$, and
10. the expected pseudo-reward v for this state.

The new predicate works similarly to $bestDoM$, is also recursively implemented, but memorizes yet computed values. It also begins each iteration step with checking possible events and processing them. This can take the agent to other situations/states and even outside the region into an exit state. We omit details on this step as it is similar to the previously presented algorithm. Afterwards, the option-skeleton is processed. Although no sequences are allowed in these, this processing can take place in various stages due to possible nesting through the use of if and $nondet$. The stage is represented by the depth d (argument 8) and is initially zero. Only the values at depth zero matter.

Primitive Actions

For a primitive action a, it is verified that the action is possible in the current situation, and the policy is set to this action. The situation resulting from performing the action in the current situation, $do(a, s)$, is determined, and the reward and pseudo-reward of the last iteration $(h - 1)$ for that successor situation are looked up in the knowledge base. To the obtained reward and pseudo-reward, the reward for the actual situation is added. The probability list for the current iteration is set to $[(do(a, s), 1.0)]$. Then, if the depth is zero, the new reward, new pseudo-reward, the probability list, and the policy are saved for this state and iteration number. Usually a primitive action would not appear at depth zero. It would mean, the local MDP allowed only this one primitive action, deterministically, and in all states.

Stochastic Procedures

For stochastic procedures the implementation is as follows:

$$
\begin{aligned}
&bestDoMOpt(o,\theta,\alpha,s,h,r,\Pi,d,\pi,v) \equiv \\
&\quad \alpha = FALSE \wedge h > 0 \wedge stoch\_proc(\theta) \wedge \\
&\quad (stoch\_proc\_poss(\theta,s) \wedge \exists(\Omega,\psi) stoch\_proc\_outcomes(\theta,s,\Omega,\psi) \wedge \\
&\quad \exists(h') h' = h-1 \wedge \\
&\quad \exists(r_{list},v_{list}) bestDoMOpt\_stoch\_Aux(o,\Omega,s,h',r_{list},\Pi,v_{list}) \wedge \\
&\quad \exists(r_s,c) prolog\_function(Reward,r_s,s) \wedge \\
&\quad (stoch\_proc\_costs(\theta,c,s) \vee \neg stoch\_proc\_costs(\theta,c,s) \wedge c = 0.0) \wedge \\
&\quad \exists(b,\gamma) option(o,b,\gamma) \wedge r = r_s - c + \gamma \cdot r_{list} \wedge v = r_s - c + \gamma \cdot v_{list} \wedge \\
&\quad \pi = \theta \vee \\
&\quad \neg stoch\_proc\_poss(\theta,s) \wedge prolog\_function(Reward,r,s) \wedge \\
&\quad \pi = [\,], v = 0.0, \Pi = [\,]) \wedge opt\_update\_Optionbase(d,o,s,h,\Pi,r,v,\pi)
\end{aligned}
$$

This works very similar to the yet seen implementation of $bestDoM$, except for two details: The rewards are discounted by the discounting factor γ defined for the option, and, at the end, the iteration values (Π, r, v, π) for this situation/state s and iteration number h are added to the knowledge base. Thereby the depth is taken into account, such that no saving takes place in case d is unequal zero. More interesting, is the definition of the auxiliary predicate, which we show again only for the case where the list of outcomes has not reached its end. The other case is very similar and can be found in Appendix A.4.

$$
\begin{aligned}
&bestDoMOpt\_stoch\_Aux(o,[(\omega,p_\omega,\psi)|\Omega],s,h,r,\Pi,v) \equiv \\
&\quad \Omega \neq [\,] \wedge \\
&\quad (\exists(\rho,s') trans^*(\omega,s,\rho,s') \vee \nexists(\rho,s') trans^*(\omega,s,\rho,s') \wedge s' = s) \wedge \\
&\quad \exists(r_{tree},\Pi_{tree},v_{tree}) bestDoMOpt\_stoch\_Aux(o,\Omega,s,h,r_{tree},\Pi_{tree},v_{tree}) \wedge \\
&\quad \exists(b,\gamma) option(o,b,\gamma) \wedge \exists(r_{rest},\Pi_{rest},\pi_{rest},v_{rest}) \\
&\quad bestDoMOpt(o,b,TRUE,s',h,r_{rest},\Pi_{rest},0,\pi_{rest},v_{rest}) \wedge \\
&\quad r = r_{tree} + p_\omega \cdot r_{rest} \wedge v = v_{tree} + p_\omega \cdot v_{rest} \wedge \\
&\quad \exists(\sigma) opt\_state\_tmp(o,\sigma',s') \wedge opt\_PL\_add(\Pi_{tree},[(\sigma',p_\omega)],\Pi)
\end{aligned}
$$

Here the outcome describing program ω gets interpreted by $trans^*(\cdot)$ obtaining a new situation s'. If that is not successful, we take the effect to be impossible and proceed the projection in the current situation. The h here is already the decreased iteration number. Thus, the call to $bestDoMOpt(\cdot)$ returns the values from the last iteration. This call in general returns immediately as the values were saved during the previous iteration. Following, the expected rewards are calculated as usual. The last line, however, is special for this algorithm: For the situation reached as an effect of this outcome (s'), the corresponding state (σ') is determined and the probability list $[(\sigma', p_\omega)]$ is created, solely containing an entry for this state together with the probability p_ω for this outcome. $opt\_PL\_add(\Pi_1,\Pi_2,\Pi_3)$ "adds" two probability lists: For those states that appear in both lists, the probabilities are added. All others are simply taken over into the new list. Hence, the overall returned list describes the probabilities of getting to a certain state after performing this stochastic procedure in the current state.

Conditionals (If-then-else)

The definition for conditionals is straightforward:

$$\begin{aligned} &bestDoMOpt(o, if(\varphi, \delta_1, \delta_2), \alpha, s, h, r, \Pi, d, \pi, v) \equiv \\ &\quad \alpha = FALSE \wedge h \geq 0 \wedge \\ &\quad (\varphi[s] \wedge bestDoMOpt(o, \delta_1, FALSE, s, h, r, \Pi, d, \pi, v) \vee \\ &\quad \neg\varphi[s] \wedge bestDoMOpt(o, \delta_2, FALSE, s, h, r, \Pi, d, \pi, v)) \end{aligned}$$

Where $\varphi[s]$ denotes the truth value of condition φ in situation s according to the situation calculus.

Nondeterministic Choice

For nondeterministic choices, the expectations for all alternatives have to be compared to determine which is the best choice. Again the implementation is straightforward and similar to the already seen, except, here we have to increase the depth argument. This is because all the alternatives are evaluated in the current situation/state and current iteration number. All of them would save their idea about the best policy and the expected values and probability list. However, we are only interested in the *best* alternative and, therefore, have to keep the individual branches from writing to the knowledge base.

$$\begin{aligned} &bestDoMOpt(o, nondet(\Theta), \alpha, s, h, r, \Pi, d, \pi, v) \equiv \\ &\quad \alpha = FALSE \wedge h \geq 0 \wedge \exists(d')d' = d + 1 \wedge \\ &\quad bestDoMOpt\_nondet\_Aux(o, \Theta, s, h, r, \Pi, d', \pi, v) \wedge \\ &\quad opt\_update\_Optionbase(d, o, s, h, \Pi, r, v, \pi) \end{aligned}$$

$$\begin{aligned} &bestDoMOpt\_nondet\_Aux(o, [\theta|\Theta], s, h, r, \Pi, d, \pi, v) \equiv \\ &\quad \Theta = [\,] \wedge bestDoMOpt(o, \theta, FALSE, s, h, r, \Pi, d, \pi, v) \vee \\ &\quad \Theta \neq [\,] \wedge \exists(r_1, \Pi_1, \pi_1, v_1)bestDoMOpt(o, \theta, FALSE, s, h, r_1, \Pi_1, d, \pi_1, v_1) \wedge \\ &\quad \exists(r_2, \Pi_2, \pi_2, v_2)bestDoMOpt\_nondet\_Aux(o, \Theta, s, h, r_2, \Pi_2, d, \pi_2, v_2) \wedge \\ &\quad (v_1 \geq v_2 \wedge \Pi = \Pi_1 \wedge r = r_1 \wedge \pi = \pi_1 \wedge v = v_1 \vee \\ &\quad v_1 < v_2 \wedge \Pi = \Pi_2 \wedge r = r_2 \wedge \pi = \pi_2 \wedge v = v_2) \end{aligned}$$

The iteration is repeated until the expected pseudo-reward at all states has converged. Convergence is tested by comparing the current value with the value before the last iteration. We say the value converged if the difference is smaller than a certain threshold value ϵ. This threshold can be defined by the predicate $options\_epsilon(\epsilon)$. If no threshold is set, $\epsilon = 0.01$ is taken as default in our implementation.

Creating the Model

So far, we have only computed the expected value and optimal policy π for each initiation state of the local MDP. However, the key to using this solution for global MDP planning is to possess a model of the behavior of this solution. The behavior of the solution is described by the policy. We already know the one-step transitions inside the

local MDP from the probability lists. Now, we have to compute the τ-step transitions from the inner states to the exit states, where τ is a random variable denoting the steps until reaching an exit state. That is, we want to know the probability of eventually ending in a certain exit state, when the policy is started in a certain inner state. Formally, for an initiation state s and an exit state t, these *exit probabilities* are defined as follows:

$$T^*_\pi(s,t) = T_\pi(s,t) + \sum_{s' \in I} T_\pi(s,s') \cdot T^*_\pi(s',t) \tag{5.1}$$

where $T_\pi(s,t)$ describes the one-step transition model for the local MDP when the policy π is applied. We have implemented an iterative algorithm to solve the resulting set of equations. The implementation is straightforward and can be found in the appendix, so we omit the details here. It is possible that the computed probabilities for reaching any exit state sum to less than one or even zero. In the latter case, the policy is describing a closed loop and indicates the presence of positive global rewards or negative global costs within the local MDP. Then it is likely that the best thing to do is simply to repeatedly earn these rewards and to never leave the region. This, actually does not fit the purpose of options and it is in the responsibility of the user to avoid or handle such behaviors.

Besides the exit probabilities which describe the outcomes of performing the option, we also assign costs to the options. These again depend on the state where the option is taken and are simply the expected accumulated global reward within the region when performing the option.

Creating the Option

We represent options as stochastic procedures. With the results of the above algorithms, we yet have everything we need to create a new stochastic procedure. Recall that stochastic procedures are procedures whose body is used in on-line execution, but which additionally have a model used in projection describing their effects. Further, they possess a precondition and may also have costs assigned to their execution. All this information can be computed from the solution of a local MDP creating a new option. The task is carried out by the preprocessor: The condition used to describe the initiation situations immediately forms a precondition. From the exit probabilities $T^*_\pi(s,\cdot)$, for each state s the list of possible outcomes can be generated. The expected rewards are taken over as negative costs.[24]

Also the procedure body for on-line execution needs to be constructed. This is guided by the following pattern:

$$proc(o, [\psi, while(\varphi_o, [if(\phi_1, \pi_1, if(\phi_2, \pi_2, \ldots if(\phi_n, \pi_n, [\,]) \ldots)), \psi])])$$

Here ψ is the sensing program defined for the option by $option\_sense(o, \psi)$. It is executed to establish the truth values for the discriminative conditions used to determine the current state. The condition φ_o holds if and only if the option is applicable in the current situation. This is basically the condition defined by $option\_init(o, \varphi_o)$. The ϕ_i are the conditions used in $option\_mapping(o, \sigma, \Gamma, \phi)$ that hold if and only if the system is in a situation that corresponds to the given state σ. This shows the need for the *states* $\rightarrow$ *conditions* mapping we initially required. The π_i are the associated actions of lower level for these states. Thus, what the procedure

[24] Remember that negative costs are beneficial and not punitive.

does after all, is to repeatedly sense all information needed to determine the state of the underlying local MDP, test the discriminative conditions of the states, and execute the corresponding policy.

5.2.3 Using Options

Since options are represented as stochastic procedures, they can be used the same way and all considerations of earlier sections about using stochastic procedures apply. In particular, an option of level n can also be used as an action for creating options at levels greater n. This way, one can build a hierarchy of abstraction. As shown in the related work of Sutton et al. [38] and especially Hauskrecht et al. [21], this kind of abstraction can reduce computational complexity tremendously. However, in their approaches, both fix the way how options are used. While Sutton et al. consider their potential to speed up the convergence of value iteration when used in addition to primitive actions, Hauskrecht et al. investigate different possibilities of combining options and primitive actions and finally focus on state space abstraction based on options. Their claim is that only with state space abstraction, options can deploy their full impact as computational leverage. This is due to the applied iterative algorithms which are based on enumerating all states. Thus, in order to achieve the desired speed-up, the state space has to be explicitly reduced before applying these algorithms. However, in our context we always work on infinite state spaces using an algorithm that "explores" that part of state space that is relevant for the computation, exploiting the knowledge about the starting point of the problem (cf. Section 5.1.3). It turns out, that this algorithm works very nicely together with options. The state space abstraction possible through options, can be seen to be done "on-the-fly" when using an option in the global problem. That in particular means that we do not have to (and even cannot) explicitly commit to some state space abstraction in advance, but still gain from the implicit state space abstraction provided through options.

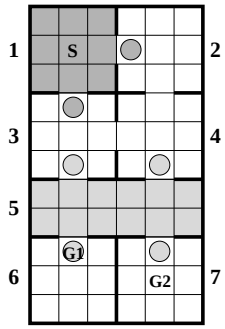


Figure 5.4: The Maze66 example environment. For the options of rooms one and five the initiation regions and the exit states are marked. Note that an agent that aims at leaving a room through a certain door may accidentally "drop-out" of the wrong door due to the nondeterminism of the performed actions.

Example 5.2.1 *Hauskrecht et al. [21] consider different ways of combining options with primitive actions creating different kinds of MDPs. By example in the Maze66 environment, we want to briefly show how easily these MDPs can also be modeled by* READYLOG *procedures using options.*

For every room-door combination an option has been created. That is, for example, for room one there are two options: one for leaving the room to the east, one for leaving to the south. These options have been created from solving local MDPs where the pseudo-rewards for these exit states have been set accordingly (high positive for the desired door, high negative for the other). Figure 5.4 shows the initiation regions and exit states for options of rooms one and five. The task faced by the agent, is to navigate from S to a goal, which is either at $G1$ or $G2$. The dynamics are as before: in each compass direction the agent is able to move, but with probability 0.09 the move goes into a wrong direction. Every move costs 1 and at the goal there is a high positive reward.

1. augmented MDP
   The augmented MDP is the MDP resulting from adding all options to the list of possible actions at each state. This resembles the approach of Sutton and his colleagues. The augmented MDP can be represented in READYLOG *as the procedure:*

```
1 proc(augmentedMDP,
2      nondet([go_right, go_left, go_down, go_up,
3              room1_2, room1_3, room2_1, .., room7_5]))
```

   where `go_right, go_left..` *are primitive actions and* `room1_2, room1_3..` *are options for leaving room one towards room two and three respectively. The augmented MDP can find solutions for both tasks (traveling to $G1$ or $G2$ respectively). This is trivially true, since all primitive actions are available.*

2. abstract MDP
   This MDP solely plans over options, thereby ignoring primitive actions:

```
1 proc(abstractMDP,
2      nondet([room1_2, room1_3, room2_1, .., room7_5]))
```

   The advantage is, that the granularity of planning is highly reduced, since the options induce a smaller state space. However, this MDP cannot find a solution to the problem of traveling to $G2$, since $G2$ is not within the set of states used by the abstract MDP. Only exit states of options are reachable.

3. hybrid MDP
   The hybrid MDP combines the previous approaches. It generally uses options only, but in the goal region uses primitive actions. It relies on the knowledge in which region the goal is located. For the case the goal is in room seven, the procedure in READYLOG *could be:*

```
1 proc(hybridMDP,
2      if( inRoom(7),
3          nondet[go_right, go_left, go_down, go_up],
4          nondet([room1_2, room1_3, .., room7_5]) ) )
```

   Note, that this procedure also forms a legal option-skeleton of level 2 (assuming the `room..` *options are of level 1). Thus, we could also create a new option using the hybrid MDP as local MDP. If the region the goal is in is known, this kind of MDP also finds a solution for any goal position. However, for each goal region, a special hybrid MDP has to be designed.*

5.2.4 Problems and Restrictions

Many open questions and problems for using options in realistic applications remain. The decomposition of the task into sub-problems whose solutions form the set of options, is crucial for the benefit of the approach. However, to the best of our knowledge, all research on the use of options base on handcrafted decompositions. Although automatic decompositions are imaginable, it is not clear how to measure the quality of a decomposition. Recent work by Amir and Engelhardt [3] present ideas for automatic decomposition in classical planning. In their approach they reduce the task to tree decomposition in graphs. They apply an approximative algorithm to solve the problem of finding an optimal tree decomposition which is NP-hard. Their ideas could be of interest for future work.

Another problem is that of finding *related* MDPs. As said, the main intention for options is to reuse them to solve several related MDPs, which would justify the overhead of creating solutions for the local MDPs. However, the task of finding related MDPs remains a current research topic. While Hauskrecht and his colleagues [21] approach this issue by their suggestion of an hybrid MDP, which only accounts for changing places of the goal in a certain region, they require that the remaining parts stay unchanged. However, in realistic domains often certain properties of the world change that in theory do not affect the applicability of an option. Nevertheless, if these changes were not anticipated during option construction, the option would in general be decided to not be possible. Thus the task is to find a measure of relatedness. In recent work, Barto et al. [32] investigate homomorphisms between semi-MDPs and show applications of that research in the options framework. There they use homomorphisms between two or more parts of a decomposition, in their example several similar rooms, and solve the corresponding local MDPs only once for some sort of abstract local MDP. By exploiting the homomorphism they can then apply the solution to all related rooms. This way or similar, one can allow reusing an option at different places in state space. So far in our and other systems, options are only applicable in the exact subset of state space where they have been defined.

Another problem of options defined by a local MDP of that kind is that the local MDPs are bound to finite state and action spaces. However, most realistic domains and also, for example, ROBOCUP Simulation League have continuous state spaces and, due to action arguments with continuous domains, also continuous action spaces. To make options still applicable, state and action abstraction would be required. That is, on top of the basic actions, higher level actions had to be handcrafted or provided through the user by other means, such that the induced state space would get at least countable. While this already contradicts to the intention of options of *automated* abstraction, it even can make options superfluous for the task, namely if by the handcrafted abstraction the problem is yet so much simplified that the remaining problem can easily be solved without applying options. Yet, there are approaches to automatic abstraction, such as described by Boutilier et al. [7].

5.3 Preprocessor and Implementation

The implementation of a preprocessor was motivated by experimental results with an early approach to integrating decision-theoretic planning into ICPGOLOG. The interpreter was in the order of a magnitude slower than DTGOLOG in solving simple planning tasks in the grid world example domains. Also in the ROBOCUP Simulation

League the time consumption for projection of simple example tasks like a goal kick were unsatisfactory.[25]

These early approaches were inspired by the projection fashion of ICPGOLOG (which is that of PGOLOG). Investigations brought to light that the main slowing-down factor was the evaluation of conditions and the therein required evaluation of expressions. As a special case, the effect axioms (`causes_val(Action, Fluent, NewValue, Condition)`) (cf. Section 4.1.2) that were used in ICPGOLOG in place of successor state axioms made fluent evaluation rather slow compared to successor state axioms of, e.g., DTGOLOG. Roughly, the increased speed of DTGOLOG is because the user implements successor state axioms and many other things directly in Prolog. The conditions in ICPGOLOG, like the ones included in the effect axioms, are formulated in a meta-language (cf. [22]) and in particular are completely independent from Prolog. While this is supposed to be more convenient for the user, it requires more time to interpret. Here the preprocessor comes into play.

The preprocessor takes a list of READYLOG files as input and processes them sequentially creating a Prolog source file as output. In doing so, it already uses the results for one file when processing the next. This is needed to allow hierarchies of options. The general philosophy for the preprocessor is to anticipate as much interpretation as possible. The core ability is to compile conditions ($\hat{=}$ formulas of the situation calculus) together with any included expressions ($\hat{=}$ terms of the situation calculus) to Prolog code which is faster in execution. With that ability, many elements of the language can be preprocessed. Furthermore, the preprocessor is used to create stochastic procedures from procedure models and computes options from the definitions of local MDPs.

Funge [17] used a compiler written in Java to preprocess GOLOG procedure to Prolog. Although the motivation for that preprocessor was equal and it does something similar, it is not related to what we present here and in particular would not fit our needs.

5.3.1 Compiling Conditions

In the ICPGOLOG interpreter [22], conditions are tested using the `holds(..)` predicate. The call `holds(C, S)` succeeds if and only if condition `C` holds in situation `S`. The predicate is recursively defined over the structure of legal conditions. Here are some examples.

```
1 holds( false, _) :- !, fail.
2 holds( true,  _) :- !.
3
4 holds( and([]),    _) :- !.
5 holds( and([P|R]), S) :- !, holds(P,S), holds(and(R),S).
6
7 holds( or([P]),   S) :- !, holds(P,S).
8 holds( or([P|R]), S) :- !, (holds(P,S) ; holds(or(R),S)).
```

Formally, compiling a condition `C` in ICPGOLOG notation to Prolog amounts to creating a Prolog predicate `P(..)` such that for all situations `S` the call `P(S)` succeeds if and only if `holds(C,S)` succeeds. The preprocessor is thus implemented in analogy

[25] Projecting a goal kick for two alternative teammates including expectations about the opponents goalie, took about 0.4 seconds which is four times longer than one simulation cycle of the soccer server. The aim should be to finish simple projection tasks like this within one cycle.

to the cases of `holds(..)`. In fact, for each such case an instance of a new predicate `process_condition(C, S, B)` is defined that returns the Prolog goal `B` equivalent to evaluating condition `C` in situation `S`. For the above cases, for example, the `process_condition` clauses are:

```
1  process_condition( false, _S, fail ).
2  process_condition( true, _S, true ).
3
4  process_condition( and([]), _S, true ).
5  process_condition( and([C|C_rest]), S, Body ) :-
6    process_condition( C, S, CNew),
7    process_condition( and(C_rest), S, C_rest_new),
8    conjunct( CNew, C_rest_new, Body).
9
10 process_condition( or([C]), S, CNew ) :-
11   process_condition( C, S, CNew).
12 process_condition( or([C|C_rest]), S, Body ) :-
13   process_condition( C, S, CNew),
14   process_condition( or(C_rest), S, C_rest_new),
15   disjunct(CNew, C_rest_new, Body).
```

The predicates `conjunct(A,B,Z)` and `disjunct(A,B,Z)` are auxiliary predicates that conjoin, respectively disjoin two Prolog goals simplifying where possible. For example `conjunct(true, B, B)` and `conjunct(false, _, false)`.

A central role in conditions play comparisons (`=, <, >, =<, >=`). To establish their truth values, the expressions of both sides first have to be evaluated. This is realized by the predicate `subf(E, V, S)`, which evaluates expression `E` to expression `V` in situation `S`. Also it is possible that conditions consist of only a single relational fluent. This also would be evaluated by `subf(..)`.

Evaluating Expressions

Roughly, `subf(..)` has three major tasks: evaluating fluents at the actual situation, evaluating functions, and evaluating arithmetic expressions. Special cases exist such that `subf(..)` leaves variables, numbers, and constants unchanged.

Again we wanted to create preprocessor rules to compile `subf`-evaluations to Prolog.[26] In fact, major parts of such expressions can already be evaluated off-line. Then, in on-line mode the actual fluent values can just be plugged in leaving only minimal remaining computations in Prolog.

The counterpart to `subf(..)` in the preprocessor is the predicate `process_subf(E, V, Body, Type, S)`, where

- `E` is the expression to evaluate,
- `V` is the representation of the evaluated expression,
- `Body` is the remaining Prolog code that has to be run on-line to evaluate the expression to `V`,
- `Type` states an only internally used type of `V` which is used to detect syntax errors and to detect sub-expressions that can at compile time already be completely evaluated, and

[26] The alternative would be to always evaluate expressions completely on-line by simply adding a call to `subf(..)` where needed in the compiled Prolog goal.

- `S` is the *formal* situation argument, which at run-time gets substituted for the actual situation.

Let us now briefly describe how the three mentioned major tasks are carried out by `process_subf(..)`.

Fluents can only be evaluated at run-time, when the actual situation is known. Thus, the preprocessor can do nothing more than add the necessary evaluation calls (`has_val(Fluent, NewVar, S)`) with a new variable (`NewVar`) to the Prolog goal (`Body`) and set this variable as the representation for the evaluated expression (`V=NewVar`). In advance, any possible arguments of the fluent are evaluated, possibly adding further goals to `Body`. One distinction has to be made for continuous fluents[27]: for these the fluent `start`, holding the current time, has to be evaluated in on-line mode as well, and the fluents value has to be determined based on that. Thus the Prolog body for these will contain something like: `has_val(Fluent, ConVal, S), has_val(start, Time, S), t_function(ConVal, V, Time)`.

Functions can be recursive. This simple fact prevents a direct evaluation of functions where they are used. Instead, each function has to be compiled to an individual Prolog predicate (`prolog_function(..)`), which we describe in the next section. After this is done, the evaluation of a function reduces to evaluating all arguments and adding a call for the corresponding predicate (`prolog_function(.., S)`) to the `Body`.

Compiling *arithmetic expressions* is more interesting. Here, on-line computations can be saved by anticipating the evaluation of parts of the arithmetics. For the four supported operators ("+", "*", "-", "/"), the `process_subf(..)` rules look almost the same (here is the example for multiplication):

```
1  process_subf(A*B, C, Body, Type, S) :-
2    local_var([ValA,ValB]),
3    process_subf(A, ValA, BodyA, TypeA, S),
4    process_subf(B, ValB, BodyB, TypeB, S),
5    process_subf_aux( *, ValA, BodyA, TypeA, ValB, BodyB,
6                      TypeB, C, Body, Type).
```

Here, `local_var(L)` creates a fresh variable for each entry of the list `L`.[28] Afterwards, the two operands `A`, `B` are processed returning their new representation, any remaining Prolog goals that need to be evaluated on-line, and their internal type. The main task of combining these results is carried out by the auxiliary predicate, which forms a key element to the utility of the preprocessor. Therefore, we will explain its functioning a little bit more detailed.

```
1  process_subf_aux( Op, ValA, BodyA, TypeA, ValB, BodyB, TypeB,
2                    ValC, BodyC, TypeC) :-
3    Expression =.. [Op, ValA, ValB],
4    (
5      /* best case: can be evaluated at compile time */
6      TypeA = eval, TypeB = eval ->
7      ValC is Expression,
8      BodyC = true,
9      TypeC = eval
10   ;
11     /* if cannot be evaluated right now: leave ValC
```

[27] Recall that continuous fluents change their value continuously over time.

[28] This is realized by the ECLiPSe library `var_name`.

```
12      uninstantiated and put evaluation into Body */
13      (
14        metaType(TypeA, meta_number), TypeB = eval ->
15        concat_string(["(",ValB,")"], StringB),
16        Expression2 =.. [Op, ValA, StringB],
17        conjunct( BodyA, (ValC is Expression2), BodyC)
18      ;
19        TypeA = eval, metaType(TypeB, meta_number) ->
20        concat_string(["(",ValA,")"], StringA),
21        Expression2 =.. [Op, StringA, ValB],
22        conjunct( BodyB, (ValC is Expression2), BodyC)
23      ;
24        metaType( TypeA, meta_number),
25        metaType( TypeB, meta_number) ->
26        conjunct( BodyA, BodyB, Tmp1),
27        conjunct( Tmp1, (ValC is Expression), BodyC)
28      ),
29      TypeC = number
30    ;
31      /* error case */
32      printf("** error in process_subf_aux: TypeA %w
33             \tTypeB %w\n", [TypeA, TypeB])
34    ).
```

Overall, there are four cases that we distinguish here depending on the types of the operands:

1. Both operands are fully evaluated. This could for example happen when the operands are constant numbers. Then also the result for this operation can be evaluated yet (`ValC is Expression`), the returned type is `eval`, and the Prolog body of goals left for on-line interpretation is empty (`Body=true`). This is the ideal case where everything can yet be evaluated off-line.

2.+3. Case two and three are symmetric. It is when one of the operands is yet evaluated, but the other is not. Then only for the unevaluated operand there remains Prolog code for on-line evaluation.

4. The remaining and worst case is when neither operand can be evaluated completely at compile time. Then the Prolog bodies have to be conjoined.

In any case, the type of each operand has to either be `eval`, marking yet evaluated numbers. Or it has to be something that still can get a number, that is, a variable, a fluent, or an arithmetic sub-expression. This sorts out ill formed arithmetic expressions like `V = 34 + noflu` where `noflu` is not a fluent.[29]

All legal combinations of types (cases 1-4) can be found in the following example:

```
function( example, V,
          V = 2 * 5 - distance(ball, oppgoal) + posX - 1)
              1.          3.                    2.
                  4.
```

[29] Note that READYLOG, as ICPGOLOG, has no sophisticated type concept for fluents. Thus, it is in general not possible to be sure that a fluent represents a number. Adding a type concept, could be done in future work.

where `distance(A,B)` is a function and all other names are fluents.

5.3.2 Processing Elements of the Language

Being able to compile conditions and expressions, many elements of the language can be preprocessed without much additional effort. Functions (`function(Name, Value, Cond)`) and action preconditions (`poss(Action, Cond)`) are straightforward to process. In both cases the involved condition gets compiled. Additionally for functions, the `Value` has to be processed by `process_subf(..)`.[30]

Above all, the preprocessor carries out some other key tasks which we are going to describe next.

Generating Successor State Axioms from Effect Axioms

ICPGOLOG uses effect axioms to describe how actions change the value of fluents. These are formulated by the user through clauses of the form `causes_val(Action, Fluent, NewValue, Cond)`. For regression (cf. Section 4.1.2) this means that the included conditions have to be interpreted every time the axiom is applied. The reason why ICPGOLOG, distinct from other GOLOG interpreters, uses effect axioms instead of successor state axioms, is because of the progression algorithm. For that algorithm such an action oriented description of effects is preferable over the fluent oriented successor state axioms.

The preprocessor transforms these effect axioms to successor state axioms applying the algorithm which was used by Ray Reiter in his solution to the frame problem [33], and which we described in Section 4.1.1. Thereby the conditions (`Cond`) and value expressions (`NewValue`) are compiled.

Although, preprocessing here comes with no disadvantages, except for a small amount of time needed for compilation, it considerably speeds up the regression. To compare the speed of regression with non-compiled effect axioms (`causes_val`s) and compiled successor state axioms, we have implemented this tiny example procedure:

```
proc( act(X), [ while(flu < X, inc), saytime ] ).
```

The procedure simply counts from zero up to the argument `X` by increasing (`inc`) the fluent (`flu`) and then prints the used CPU time since the beginning of the task. We have run this procedure 20 times in a row with increasing values for the argument `X`. Starting with 100, we increased at steps of 100. So in the final iteration, a situation term containing 2000 times the primitive action `inc` is operated by regression. Here is the effect axiom for action `inc` and fluent `flu` plus the resulting successor state axiom produced by the preprocessor.

```
causes_val( inc, flu, L, L = flu+1 ).
```

```
1 ssa( flu, PreVar0, [inc|Srest] ) :-
2             !, (has_val(flu, PreVar4, Srest),
3             PreVar3 is PreVar4 + (1)),
4             PreVar0 = PreVar3.
5
```

[30] This is because `Value` itself can be an expression (instead of just a variable) like, for example, in `function(f, V+3, V = flu*2)`.

```
6  ssa( flu, NewValue, [_UnknownAction|Srest] ) :-
7              !, has_val(flu, NewValue, Srest).
```

Figure 5.5 shows the chart comparing the time consumption when the task is run with effect axioms and when run with successor state axioms. Please note that the y-axis has a logarithmic scale. Using compiled successor state axioms in this example was

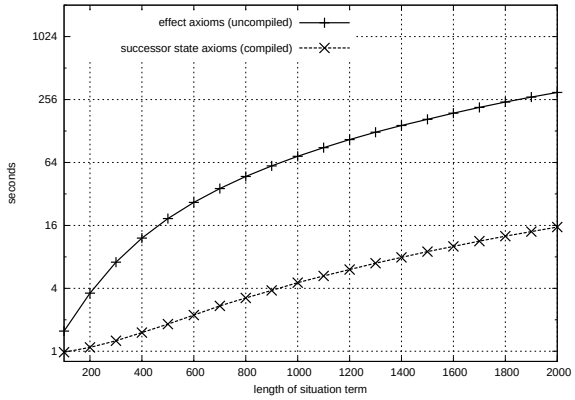


Figure 5.5: Comparing the speed of regression when using effect axioms like in ICP-GOLOG and when using the (compiled) successor state axioms produced by the preprocessor. The progression mechanism had to be disabled for this test, in order to create situations of the desired length.

about 20 times faster than using non-compiled effect axioms. The time for preprocessing was 0.01 seconds. The result is representative in the sense that only one simple condition (`L = flu+1`) was used containing only one arithmetic expression. The savings even increase when more conditions are to be evaluated and more complex arithmetics are used. Overall, this comparison supports our observations from early versions of READYLOG, which appeared to be ten times slower than DTGOLOG. This was because these early versions did not yet use the preprocessor.[31]

Processing Stochastic Procedures and Explicit Events

As described in Section 5.1.1, the user in READYLOG can define stochastic procedures by providing a procedure model and a procedure precondition for any procedure. These and optional procedure costs definitions are also transformed by the preprocessor to reduce the required on-line interpretation. Among other things, the task involves transforming occurring control constructs to Prolog. For tests (`?(Condition)`) this simply amounts to compiling the condition and conjoining it with the remaining Prolog body. For `if(C, A, B)`-constructs the pattern `( CC -> CA ; CB )` is used. Here `CC` will be replaced by the compiled condition, `CA` and `CB` are the compiled alternatives `A, B`. The remaining elements of the task are relatively straightforward to conduct, so that we skip details on these syntactic transformations. Explicit events can be processed by the same transformation rules.

[31] DTGOLOG does not need a preprocessor since the user has to implement the successor state axioms already in Prolog.

Options

We described the algorithms for automatically creating options from local MDPs along with their models in Section 5.2. These are implemented in the preprocessor. As said, the preprocessor takes a list of files as input. These are processed sequentially, where the results for one file can already be used by the next. This is needed to enable hierarchies of options. The organization of these files is intended as follows: In the first file the basic dynamics of the world are defined, such as fluents, effect axioms, functions, and primitive actions. In particular, only actions of level 0 (cf. Section 5.2) are defined here. In the next file, stochastic procedures and options of level 1 are defined, that is, procedures with procedure models that only use actions of level 0 and options which only use actions of level 0 in their option-skeletons. Similarly, later files define actions of higher levels.

5.3.3 Conclusion

As a conclusion about the preprocessor we can say that it is very useful. On the one hand, it has the potential of speeding things up by a factor between ten and 20 without any disadvantages.[32] On the other hand, the preprocessor helps us in getting independent of Prolog. It would be very easy to modify the preprocessor to create code of some other kind. Then, a READYLOG interpreter could be implemented in other, potentially faster, programming languages. The crucial first step would have to be to implement the situation calculus in that language.

[32] We ignore time for preprocessing since it can be done off-line and even then takes negligibly little time.

Chapter 6

Experimental Results

In this chapter we will present some experimental results made with the new READYLOG interpreter. First, we present results on options in a grid world domain. Then, we consider the ROBOCUP Soccer Simulation and mobile robotics, where the results for mobile robotics are taken from our participation at ROBOCUP 2003 Mid-Size League World Cup at Padua, Italy.

6.1 Grid Worlds

We have taken over the Maze66 environment used by Hauskrecht et al. [21]. In first place, we use this domain to illustrate the potential of options and show their seamless integration into READYLOG. In [16], we extended the original DTGOLOG with options and reported results made in this environment. We have run the same tests with the new interpreter, in order to compare its performance to DTGOLOG.

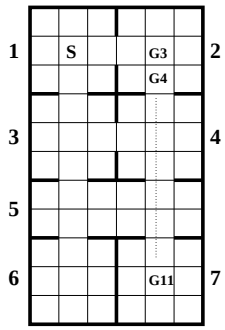


Figure 6.1: The Maze66 environment. In this domain, eight different tests were conducted: the agent always started in `S` and then aimed at reaching a certain goal `G3..G11`, which had a high positive reward. The number of the goal (3..11) is the manhattan distance from the start.

Different navigation problems were formulated: As depicted in Figure 6.1 the task was to reach a certain goal (`G3..G11`) located at different Manhattan distances from the start (`S`).

We compared three approaches to these problems which are depicted in Figure 6.2:

1. *full MDP planning*
   The agent repeatedly chooses from its basic actions `go_right, go_left, go_down, go_up`. The corresponding READYLOG program is the following:

```
proc( planning,
  [nondet([go_right, go_left, go_down, go_up]), planning])
```

   As usual all basic actions are uncertain: with probability 0.91 an action succeeds. With the remaining probability the agent will instead move in any other compass direction (if possible).

2. *heuristics*
   For each room the agent has heuristic procedures for leaving the room by a certain door. For room one the procedure for leaving towards room two is

```
1 proc( dt_room1_2,
2       [ ?(pos = [X,Y]),
3         if( Y < 1, go_down,
4         if( Y > 1, go_up, go_right) ) ]).
```

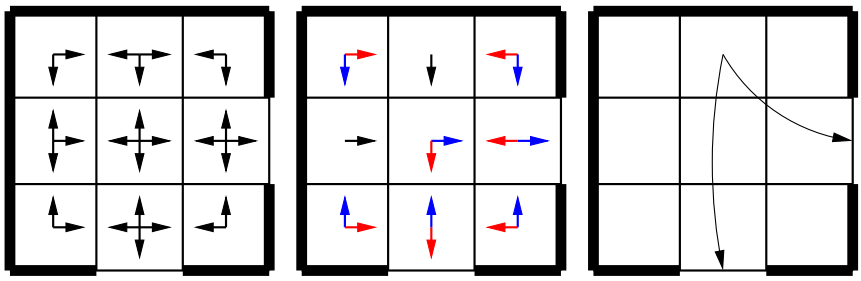
   These procedures are provided by the user and are an example of how domain knowledge can be incorporated, as was intended with the DTGOLOG approach. In the overall task the agent then chooses from these procedures as long as it is not in the goal room, where only basic actions are considered.

3. *options*
   For each room-door combination, options have been created via the definition and solution of local MDPs. For room one and the door towards room three the definition was, for example, the following:

```
1  option( room1_3,
2          nondet([go_right, go_down, go_left, go_up]), 0.9).
3  option_init( room1_3,
4               and([ domain(X, [0..5]), domain(Y, [0..10]),
5                     pos = [X,Y], inRoom(1)]) ).
6  option_sense( room1_3, exogf_Update).
7  option_beta( room1_3, inRoom(3), 100 ).
8  option_beta( room1_3,
9               and(not(inRoom(1)), not(inRoom(3))), -100 ).
10 option_variables( room1_3, [pos]).
```

   The option-skeleton is simply the choice between all basic actions. The initiation set is the set of all situations where the x-coordinate of the position is an integer number in the interval $[0..5]$, the y-coordinate is an integer number in $[0..10]$, that is, the position is legal in our domain, and the position is within room one. All relevant fluents can be sensed by the action `exogf_Update`. Depending on the room where the region is left, the pseudo-reward differs: If after leaving the room, the agent is in room three, a pseudo-reward of 100 is given. Otherwise it is -100. The mapping between situations and states relies solely on the position.

   Using these options an hybrid MDP like the one of Section 5.2.3 (also confer Section 3.2.2 and [21]) can be used to solve the task. That is, while not in the goal room, the agent only chooses among the options defined for the current room. In the goal room the basic actions are used.



(a) full MDP planning (A) (b) heuristics (B) (c) options (C)

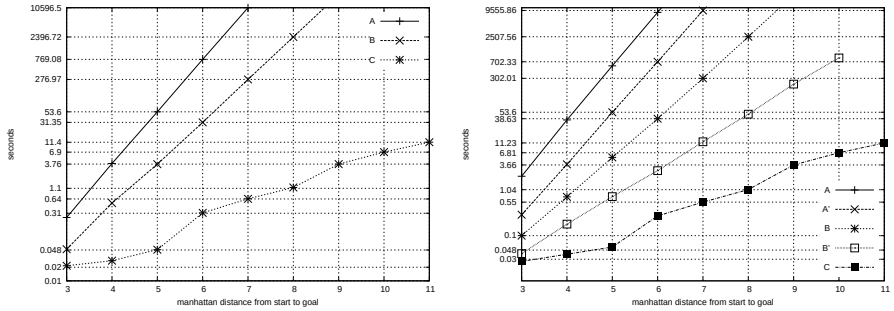
Figure 6.2: The three main approaches depicted by arrows: (a) in each state the agent can choose from all possible actions; (b) the agent only chooses from actions that lead to a door, blue arrows lead to the door to the right, red ones to the door at the bottom, black ones are common for both; (c) the agent chooses one of two options committing to an entire behavior leading to one of the doors.

While all approaches find a policy to get to the goal, the time for computation differs immensely. Figure 6.3 (a) shows the results we obtained with our extension of DTGOLOG. The problem implementations in both languages are semantically equivalent such that the results are comparable.[1] Figure 6.3 (b) shows the results obtained with READYLOG. Here, additionally we have run the first case (full planning) for comparison with precompiled successor state axioms (A') and without (A), using the effect axioms. The results compare to the observations reported at the end of the previous chapter.

Comparing the two charts, especially curve A, shows that the new interpreter is comparable in speed to DTGOLOG. This means that although we have increased expressiveness and – as we believe – user friendliness, we have not compromised performance. This would not have been possible without the preprocessor as one can see clearly from curves A and A'. Still the full planning approach here remains intractable for larger horizons.

Obviously, using heuristics provides some speed-up. This speed-up is founded in the reduction of nondeterminism: instead of choosing from all available actions, it only considers actions leaving the room by one of the doors (case B). Additionally to the above stated heuristic procedures, we have run the tests in READYLOG with a slight modification: instead of always choosing from these heuristics, this is only done once for each room (B'). This resembles the options approach. Still the computational performance is worse than for options. To understand the reason for that, we have to return to the earlier problem that infinitely many situations intuitively describe the same state. Figure 6.4 shows three different examples how room one could be left to the right. All three could happen with the heuristic procedure due to the uncertainty of the basic actions. Although some trajectories are very unlikely, e.g. the dashed one in the figure, they are all considered by the projection mechanism. As a matter of future work it could be worthwhile to find a sound lower bound for probabilities to sort out those trajectories that cannot influence the overall result anyhow (cf. Chapter 7). After

[1] The syntax of DTGOLOG and READYLOG differ.



(a) extended DTGOLOG (b) READYLOG

Figure 6.3: The resulting times using the extended DTGOLOG interpreter as in [16] and using READYLOG: (A) full planning, (B) applying heuristics, (C) using options. The curve (B') could not be drawn further due to stack overflows in the ECLiPSe Prolog system for any larger horizon than ten.

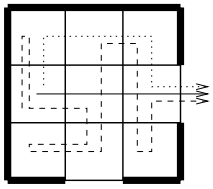


Figure 6.4: Three ways for leaving a room. Each constitutes a different situation for the same exit state.

eventually ending in the exit state in the room to the right, all these trajectories refer to different situations. As described when formally specifying the implicit MDP in DTGOLOG and READYLOG, these situations also refer to different states although this seems counterintuitive. For all these states the further projection is performed, causing more computational effort than needed.

This is exactly where options come into play. Due to the explicit mapping from situations to states and back, we can *aggregate* intuitively equal situations. In the example at hand, all possible ways would lead into the same exit state, such that from there on only one projection is pursued. This nicely shows that the *mapping* is very central to the benefit of options in the extended DTGOLOG and READYLOG. By the mapping the user defines his intuition about what characterizes a state or, seen from the other perspective, when two situations are considered equal.

Options and heuristic procedures have in common that they heuristically reduce the complexity for the cost of optimality: If the heuristics or local MDPs are badly chosen, the resulting *solution quality* can decrease, that is, for example, the length of the found path can increase. In general, it is at the responsibility of the user to provide good heuristics and only the full planning approach can always guarantee to find the optimal solution. Nevertheless, in on-line scenarios, the required time for finding a policy has usually to be included in the quality criterion. Then, solutions using options

and heuristics can even be considered better than the "optimal" but slow full planning solution. In the example, this becomes obvious when taking the time the agent needs in getting to the goal as the performance measure and assume that each action takes one second to perform. Then, the agent would rather take a sub-optimal trajectory to the goal provided by the options approach instead of waiting a long time for the optimal policy.

With regard to the solution quality, the comparison is complicated: In the presented examples, the solution of the options approach was even better than that of all others including the full MDP planning approach.[2] This is because we are not using value iteration at the global scope to solve the problem, but use decision-tree search with a fixed horizon. In the tests, we always chose the minimal horizon needed to find some path to the goal. That inplies that if this path was accidentally left, the agent would get lost. To obtain results equal or better in quality to that of the options approach, we would have to run the other approaches with larger horizons, taking even more time. However, we do not intend to start a discussion on solution quality when using options and refer to the literature [21] where this issue is well investigated. These considerations equally apply to our setting. After all, it seems apparent that for dynamic on-line settings where time *is* a criterion, it is obvious that using options is the best of all described approaches.

It remains to point out that although the presented speed-ups provided by options seem tremendous, this is also because this example domain is somewhat designed for the use of options, i.e., the regional decomposition is suggestive (rooms) and the regions are only linked by relatively few connections (doors). Also the state space is small. In the next section we will see which kinds of difficulties remain to be overcome for using options in more complex problems.

6.2 ROBOCUP Soccer Simulation

Based on the overall architecture described in [14], a system for specifying the behavior of soccer players in READYLOG was used to evaluate the new interpreter in the ROBOCUP Soccer Simulation environment. This system in turn is build on top of the UvA Trilearn base system (cf. Section 2.2.2). In a nutshell, the architecture comprises a *reactive*- and a *deliberative component*, which enables quick responds to changes in the environment, but also allows to execute plans of several actions that were created by deliberating. Thereby, the deliberative component can be specified in READYLOG. Roughly, the primitive actions are the skills of the UvA Trilearn system. Recall that this system distinguishes three skill levels, forming three levels of abstraction. Depending on the level, the planning problem in the deliberative component naturally has different granularities. These granularities manifest in the length of a plan for achieving a certain goal. For a discussion on the different models of granularity see [22]. Here, we will reduce our consideration to the highest level of abstraction, where actions are taken only from the high-level skills of the base system. All tests were run on a standard PC with an 1.7GHz Pentium 4 processor and 1GB memory.

[2] Solution quality was measured by the overall expected reward for the returned policy. This is a reasonable measure for quality, as discussed in Section 3.1.

6.2.1 Comparing to ICPGOLOG

We aimed at comparing the new READYLOG interpreter, especially the performance of the included projection mechanism used in planning, to the ICPGOLOG interpreter. To do so, we implemented a goal shot scenario and a direct pass scenario similar the ones used in [22].

Goal Shot

Basically, the task was to define a procedure model for the goal shot procedure. As in [22] we take the opponent goalie and closest player to the pathway into account: For both it is projected whether they are able to intercept the goal shot. This model can be used for a stochastic procedure after it has been preprocessed (cf. Section 5.3). Planning a goal shot with that model in READYLOG takes *0.01 seconds*. Projecting the same task in ICPGOLOG took 0.35 seconds in the best of all described cases on a comparable machine (as reported in [22]).

Direct Passes

Similar results we obtained for the direct pass scenario: For the task of deciding to which of the two closest teammates to pass to, the READYLOG interpreter needs again only *0.01 seconds*[3]. This compares to 0.25 seconds in ICPGOLOG. Even for a complete team of eleven players, the decision on which of all teammates to pass to takes only 0.07 seconds in READYLOG.

These two examples show clearly the increase of speed in READYLOG compared to its predecessor ICPGOLOG. The speed-up is explained by two reasons: the applied projection mechanism and the use of the preprocessor. The way projection is done in ICPGOLOG is more flexible and possesses a richer expressiveness. However, it is neither designed nor suited for the creation of plans by solving nondeterminism.[4] A probably greater contribution to the speed-up comes from the preprocessor. This is especially suggested by the results at the end of the last chapter (Section 5.3). The preprocessor in this special case provides us with two advantages: the obvious speed-up and the independence of Prolog. In fact, for creating the results with ICPGOLOG, most of the computations were implemented directly in Prolog (cf. [22]). By that, the author did manually what the preprocessor now does automatically. If the computations had purely been implemented in ICPGOLOG functions, they would have taken even longer.

Concluding we remark that the presented speed-up overcomes the barrier towards realistic applicability in the ROBOCUP Soccer Simulation within the initially described architecture. The soccer server accepts player commands every 0.1 seconds. If during one such cycle no command is sent, no action is executed by the player. It turned out that teams are only competitive if they are able to use every cycle for setting commands. While the described hybrid architecture helps in this regard, it seems still necessary that simple decisions, like whom to pass to, are made within one cycle. We can now meet this requirement.

[3] This is the the lowest amount of time measurable by the ECLiPSe Prolog system.

[4] In fact, it cannot even handle nondeterminism as this was not an issue in the design.

6.3 Mobile Robotics: ROBOCUP Mid-Size League

We have used READYLOG for specifying the high-level behavior of our soccer-playing, physical robots at the ROBOCUP 2003 Mid-Size League World Cup at Padua, Italy.

6.3.1 Problem Implementation

In the realization we used the skills provided by the skill module as the basic actions (cf. Section 2.3.2). The most relevant of these were:

- *goto_global/goto_relative*
  Drive to a given global position or to a position given relative to the robot.
- *turn_global/turn_relative*
  Turn to a given global angle or to an angle given relative to the current robot angle. If the ball is close in front, it will be pushed to the side to a distance depending on the turn speed which again depends on the turn angle.
- *dribble_to*
  Having the ball, go to a certain position. If the ball is lost – it is not anymore in front of the robot – the action fails.
- *guard_pos*
  Maintain a guarding position on the line between the own goal and the ball.
- *intercept_ball*
  Intercept the ball. This usually only succeeds if the ball is standing still.
- *kick*
  Trigger the kicking mechanism. If the ball is close enough, it will be accelerated to the front.
- *move_kick*
  Move towards a given position, and as soon as the angle to that position is below a certain threshold kick. This enables the robot to kick into directions it is not currently heading.

These actions were implemented as stochastic procedures. The procedure body sends the appropriate command to the skill module and waits until that reports the completion (success or failure) of the action. The procedure model describes our intuition about the effects of the action. Here is the example for the global turn action, which is assumed to be always possible:

```
1  proc( turn_global(Own, Theta, Mode),
2        [ send(nextSkill(Own), do_skill_turn_global(Theta, Mode)),
3          ?(nextSkill(Own) = nil)
4        ]).
5
6  proc_poss(turn_global(_Own, _Theta, _Mode), true).
7  proc_model(turn_global(Own, Theta, _Mode),
8    [ if(isKickable(Own),
9         /* turn will affect the ball */
10        [?(and([AgentPos = agentPos(Own),
11                AgentAngle = agentAngle(Own),
```

```
12                AgentPos = [X,Y],
13                TurnAngle = Theta - AgentAngle,
14                YDiff = TurnAngle*abs(TurnAngle) / 3.4,
15                geom_PosRel([X,Y,AgentAngle],[1,YDiff],NBPos)])),
16         set_ecf_local_ballPos(NBPos,[0,0])],
17        /* else: turn will NOT affect the ball */
18        []
19        ),
20     set_ecf_agentAngle(Own, Theta, 0)
21   ]).
```

The model describes the intuition that if the ball is close to the front (`isKickable(Own)`) of the acting robot (`Own`), the ball will be pushed in the direction of the turn. The distance (`YDiff`) depends on the turn angle. This distance is the most heuristic element in this model and is simply based on real world observations. We will describe the models for the other actions when needed.

Furthermore, all relevant and accessible world information is available to the agent: the own estimates of the robot's own position, position of the ball, and the positions of the opponents, to name the most important ones. Additionally, data from the control computer like the fused position of all ball estimates and the current play-mode is available.

We have described the actions. The transition model is defined by the action models, and the state space, as always, is the set of all situations. What still is missing for an MDP are the rewards. We define the reward for a situation based on the position and velocity of the ball. Positions inside or in front of the opponent goal get high positive rewards, while on the other hand ball positions inside or close to our goal are assigned highly negative rewards. If the ball is moving, the position to be evaluated is taken as the intersection of the ball trajectory with the boundaries. Otherwise, the ball position itself is used to evaluate the situation.

6.3.2 Agent Behavior

While the goalie was controlled without READYLOG in order to maintain the highest possible level of reactiveness, all other players of our team had an individual READYLOG procedure for playing. We assigned fixed roles to the three field players: defender, supporter, and attacker. In general, the defender keeps a guarding position between the own goal and the ball (`guard_pos`), the supporter maintains a position at medium distance to the ball to stand available for taking over the ball from the attacker, and the attacker cares about taking the ball into the opponents goal. However, often during play, also defender and supporter might be in a good position for carrying forth the ball. Therefore we have realized some sort of *dynamic role switching*. The function `bestInterceptor_All` returns the number of the player which is considered the best for going for the ball. This is used by all players to decide whether to follow their usual role or to carry out some form of attacking behavior. If the attacker believes it is not the best interceptor, it switches to a supporting behavior.

Deliberation is at any time only performed by an agent that believes to be the best interceptor. This has the advantage that the behavior of all other players is deterministic and thus easier to predict for the planning agent. We here present the planning task the attacker solves being the best interceptor. The task is described by the following `solve`-statement (cf. Fig. 6.5):

```
1  solve(nondet(
2    [kick(ownNumber, 40),
3     dt_dribble_or_move_kick(ownNumber),
4     dt_dribble_to_points(ownNumber),
5     if(isKickable(ownNumber),
6        pickBest(var_turnAngle, [-3.1, -2.3, 2.3, 3.1],
7          [turn_relative(ownNumber, var_turnAngle, 2),
8           nondet([
9             [intercept_ball(ownNumber, 1),
10             dt_dribble_or_move_kick(ownNumber)],
11            [intercept_ball(numberByRole(supporter), 1),
12             dt_dribble_or_move_kick(numberByRole(supporter))]
13          ])
14         ]),
15       nondet([
16         [intercept_ball(ownNumber, 1),
17          dt_dribble_or_move_kick(ownNumber)],
18         intercept_ball(ownNumber, 0.0, 1)])
19       ),
20    intercept_ball(ownNumber, 0.0, 1)
21   ]), 4)
22
23 proc(dt_dribble_or_move_kick(Own),
24      nondet([
25        [dribble_to(Own, oppGoalBestCorner_Tracking, 1)],
26        [move_kick(Own, oppGoalBestCorner_Tracking, 1)]])).
27
28 proc(dt_dribble_to_points(Own),
29      nondet([
30        [dribble_to(Own, [2.5, -1.25], 1)],
31        [dribble_to(Own, [2.5, -2.5], 1)],
32        [dribble_to(Own, [2.5, 0.0], 1)],
33        [dribble_to(Own, [2.5, 2.5], 1)],
34        [dribble_to(Own, [2.5, 1.25], 1)]
35      ])).
```

The set of alternatives made up from this plan-skeleton is best described by Figure 6.5. Having the ball, the agent can always think of a straight kick (`kick(ownNumber, 40)`) or to `dribble` or `move-kick` towards the goal (`dt_dribble_or_move_kick(Own)`). The function `oppGoalBest-Corner_Tracking` determines the position of the opponents goalie in the goal and thereby decides which is the better corner to shoot at (in the picture it is the right corner). Also the robot may dribble to one of five predefined positions on a line in front of the opponents goal (`dt_dribble_to_points(Own)`).[5] Additional alternatives exist depending on the position of the ball: if the ball is in kickable range, the robot considers the possibility to turn to one of four angles, pushing the ball to the left or right. The ball can then either be intercepted again by the attacker himself and carried to the goal or the supporter can do that. Here, the static behavior of the supporter is taken advantage of. The attacker assumes the supporter at its side, so that he can *pass* the ball to him. This behavior was thought of for getting past opponents blocking a

[5] This function is an ideal example where using `pickBest` would be more convenient for the user. However, for performance it does not make a difference to the `nondet`-expression used here instead.

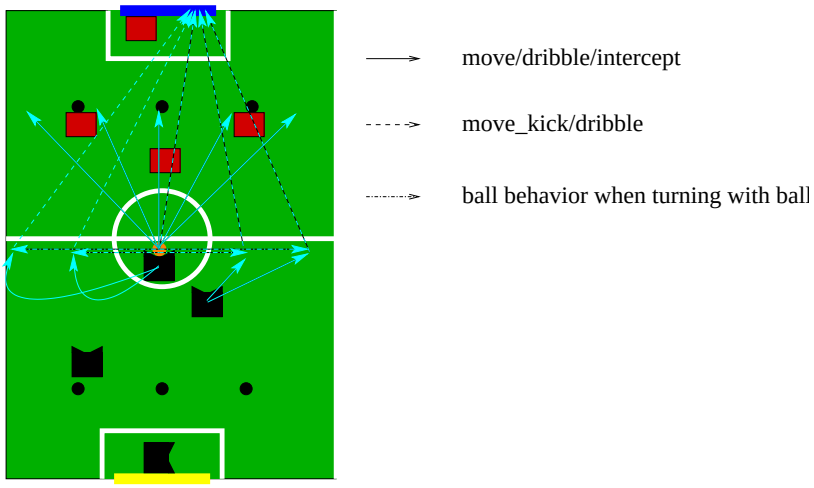


Figure 6.5: The set of alternatives for the attacker when he is at the ball. The red boxes denote opponents, the black ones are teammates. Everything else are field markings.

straight way to the goal.

Dribbling and intercepting the ball are modeled as uncertain. Both have a success and a failure case. Especially intercepting the ball is considered to be difficult, so that the probability for a failure is assumed to be 0.8.

In the case where the robot has the ball, the decision tree created from the users choices and natures choices has 62 leaves. This is a considerable number of alternatives for ROBOCUP, but should not be compared with planning tasks in simplified grid world examples. In those examples, trees with a lot more leaves are used, and still solutions are found more rapidly (time consumptions for the present ROBOCUP example are investigated in the next section). This is, because the applied models connecting the nodes of the tree there, are much simpler and thus require less computational time.

6.3.3 Experiences at ROBOCUP 2003

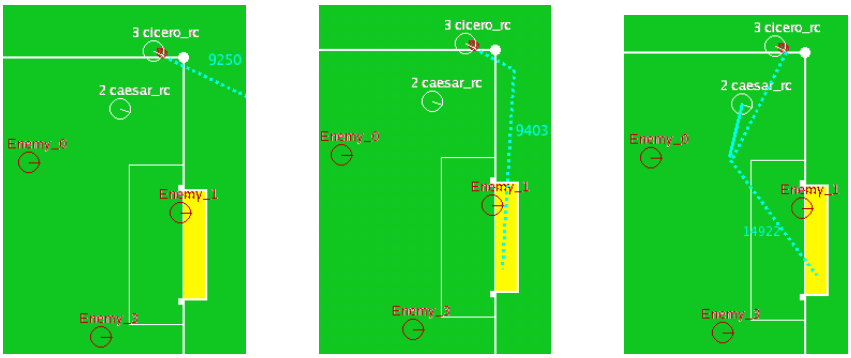
Let us consider an example situation that occurred during a match against the team of *ISePorto* in front of their goal. Figure 6.6 shows the view of the world as logged during the game. This means, the positions of the ball (in the upper right), our robots (in white), and the opponents (red) are estimated based on the fused information from all of our robots.[6] In this situation most of the dribbling actions are considered impossible: The robot has only limited control over the ball. In particular, it is not possible to drive in circles with low radius without loosing the ball. Thus, we set the precondition for dribbling as depending on the angle to the target point. In this situation the attacker (Cicero) considered, among others, the following alternatives, which are illustrated in Figure 6.7[7]:

- *a straight kick*
  (Figure 6.7 (a)) Kicking in this situation would shoot the ball out of bounds. The

[6] We remark that these positions in general are quite erratic due to time lags in the remote connections and due to sensor inaccuracies. This is another source of difficulties for decision making.

[7] All presented data is taken from log-files that were recorded during the game.

Figure 6.6: A situation that occurred during a match agains *ISePorto*: One of our robots (Cicero) has the ball and the supporter (Caesar) is at its place.



(a) (b) (c)

Figure 6.7: (a) a straight kick (b) a move-kick towards the empty opponents goal corner (c) team-play

associated reward is based on the intersection with the goal line and is computed as 9250.[8]

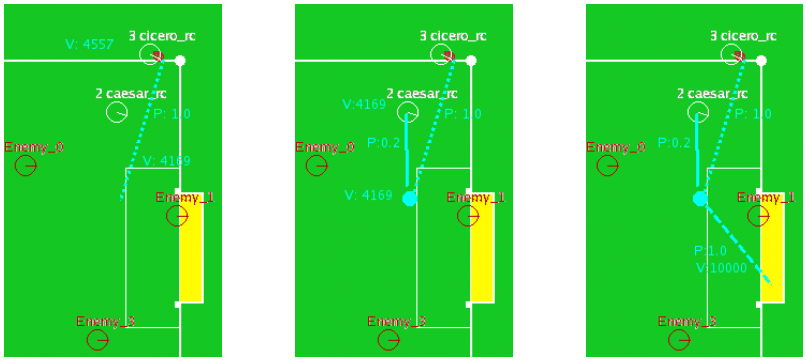
- *a move-kick to the goal*
  (Figure 6.7 (b)) The best corner to shoot at is obviously the right one. However, the model describes a move-kick by driving one meter to the front and then accelerating the ball towards the target.[9] Again the ball would go out of bounds,

[8] We only use integer numbers here, to improve readability.
[9] Of course, this model again is very heuristic and is based solely on observations during testing.

achieving a similar reward as the kick (9403).

- *team-play*
  (Figure 6.7 (c)) The action eventually taken was to turn with the ball, pushing it in front of the supporter.

Let us see what lead to this decision. For readability we only use integer values. The initial situation has a reward of 4557 and turning with the ball will certainly (probability = 1.0) lead to a situation with reward 4169 (see Figure 6.8 (a)). Next, the supporter (Caesar) is assumed to intercept the ball. This is only successful with probability 0.2. The reward for the new situation is unchanged, since the ball is not moved – in this model (see Figure 6.8 (b)). However, the intercept has negative costs, as the ball is not far for the supporter and it is in front of the opponents goal, where an intercept always seems a good idea. Alternatively, interception by the attacker is considered. But since that player is farer away from the ball, the costs for that are higher (see Figure 6.9). Finally, the supporter now having the ball is assumed to score a goal by performing the move-kick skill (see Figure 6.8 (c)). The resulting situation, with the ball in the opponents goal, gets a reward of 10000. The move-kick also has negative costs of 70. This is to encourage the robot to generally shoot more often. Overall an expected reward of 14922 is computed for this policy. Figure 6.9 shows the resulting decision



(a) (b) (c)

Figure 6.8: Team-play

tree for the three described alternatives. The optimal policy is marked by a thick line. The real decision tree was much larger (about factor 4). We left out the other possible turn angles (appearing at line 6 of the source code on page 93) which created more branches of the kind considered. Overall decision making in this situation took 0.61 seconds.

In general, the time the attacker spent on projection depended highly on the number of alternatives that were possible. The most significant difference it made whether the ball was kickable or not (all times in seconds):

	examples	min	avg	max
without ball	698	0.0	0.094	0.450
with ball	117	0.170	0.536	2.110

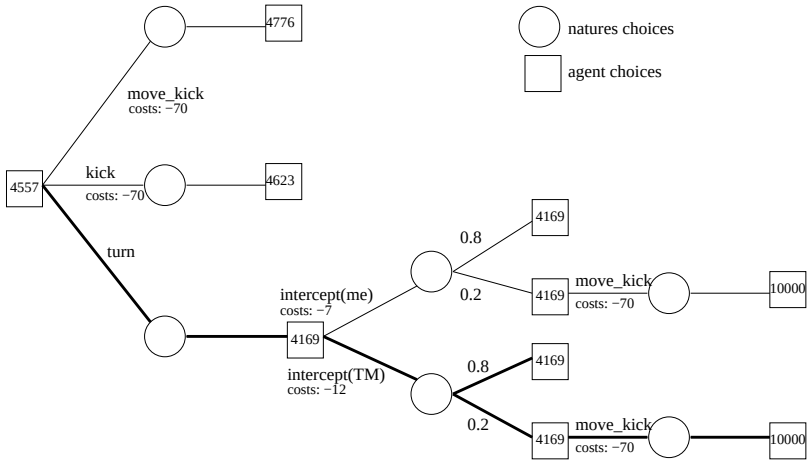


Figure 6.9: The decision tree traversed by the optimization mechanism in this situation.

These times are worse than the earlier seen numbers from the Soccer Simulation. The speed difference is mainly explained by three facts: Firstly, the processor of the computers in the robots runs at about half the speed of the ones where the simulation results where produced (Pentium 3, 933MHz vs. Pentium 4, 1.7GHz). Secondly, the robot was running several other programs consuming computation power. Thirdly, the applied models and the reward function were more complex. Especially the latter is crucial as it gets computed at each situation visited in projection. From experience we can say that nearly all the time is spent for determining successor situations, that is, doing projection, and reward calculation. Nevertheless, the presented amounts of time are still small enough for the Mid-Size League as this league is not yet as dynamic as the simulation.

A key insight of the soccer experience with the interpreter is that the action models and the reward function are very crucial. The user is recommended to take great care designing them. Figure 6.8 (a), for instance, reveals a “mistake” in the used reward function: Here it appears much better to have the ball on the penalty area line in front of the opponents goal than at the touch line. Unluckily, this opinion had not been modeled into the reward function, such that the touch line position is evaluated higher for being farer into the opponents half. The agent, which is almost exclusively guided by the reward function, is very sensitive to such mistakes. They can easily lead to unintended behavior. All the more, sound approaches to creating reward functions, for example by learning, could be beneficial and investigated as future work.

Results at ROBOCUP 2003

This approach enabled us to set up a competitive team. In the first round robin we won all games except one versus the former world champion which we lost. Unluckily, in the second round robin we missed reaching the quarter finals by one goal. Furthermore, we ended second in the technical challenges with only about half an hour of preparations. We believe that this speaks for the flexibility of our approach.

The problems preventing more successful playing did not seem to stem from the interpreter itself. Some problems were induced by low-level tasks like localization, ball

detection, and collision avoidance and by hardware issues. But also in high-level control there still seems to be much room for improvements: as mentioned the reward function and models are crucial. These should be generated in a more sophisticated fashion, for example with the help of learning techniques. Also it got apparent in play that the interaction between the robots needs some review, to say the least. It frequently happened, for example, that the discriminative function (`bestInterceptor`) seemed to be flipping. In such situations two robots could not decide on who should take the ball. Crucial functions like this must be made more stable. Also for team-play behaviors like the one planned by the attacker in the example often fail in practice, as other players involved in such a plan do not act as were expected. Although these problems are not directly linked to the interpreter itself, in future work one should still keep an open mind on how extensions of the interpreter could possibly support finding solutions here.

Chapter 7

Conclusion and Future Work

We have integrated the idea of DTGOLOG to combine explicit agent programming with MDP planning into ICPGOLOG creating the new language READYLOG. This included the development of a new approach to on-line decision-theoretic planning which we compared to an existing approach by Soutchanski [37]. Our approach includes the possibilities of the other approach but adds to it substantially. It allows for exogenous actions and offers more freedom in defining uncertainty. We have revealed a shortcoming in the other approach when operating in highly dynamic domains where frequent sensing is a must. Also we have tackled the problem of when to break policy execution, that is, when to consider a generated plan to be invalid. This problem had not been treated in the other approach.

Moreover, we have extended our approach by the concept of options to enable an exponential speed-up of planning where applicable. We support the automatic creation of options from the solution of local MDPs and integrated them seamlessly by compiling them to stochastic procedures. In particular, we offer to define options over options, allowing the user to abstract hierarchically with automated support.

Based on the ICPGOLOG interpreter we have developed an interpreter for READYLOG implemented in ECLiPSe Prolog [10]. To improve user-friendliness and especially to improve on-line performance of the projection mechanism, we have further implemented a preprocessor that compiles certain parts of READYLOG programs to Prolog. Tests showed that this preprocessor adds considerably to the performance, increasing the speed by about factor ten in average compared to ICPGOLOG.

We have tested the interpreter in three very different example domains to evaluate its use under different requirements. It showed that especially options were of high use in discrete and finite domains. There we also saw that the new interpreter is competitive to DTGOLOG if the preprocessor is used. In a simulation domain we tested READYLOG against ICPGOLOG and noted a considerable performance increase. Finally, in a real world environment we extensively used the interpreter to control mobile robots in playing soccer at a world cup tournament.

The performance was overall better than expected, which is mainly the merit of the preprocessor. However, it would be desirable to further reduce the time spent for planning. Unfortunately, we did not find a way to beneficially apply options in the ROBOCUP domain. While in the Mid-Size League that was also due to limited testing opportunities, in the simulation it was mainly caused by the continuous character of the domain.

For future investigations, it might be interesting to extend options so that they are

able to handle infinite and especially continuous state and action spaces. Also it should be possible to have nondeterministic actions with an infinite number of possible outcomes.[1] This would open great new opportunities for their applicability. It would then be imaginable to run automated abstraction in areas where before abstraction were laboriously conducted by hand. For example, the skill abstraction created by UvA Trilearn (cf. Section 2.2.2) could perhaps be generated semi-automatically. The uncertainty of the movement of objects, as described in Section 2.2.1, could then be easily modeled. To do so, either the applied methods for solving local MDPs would have to be extended accordingly or other methods capable of operating on infinite/continuous spaces would have to be found and used.

Another idea for speeding up the presented optimization algorithm could be the following: Imagine this situation at a user's choice-node: One branch was already projected and returned an expected reward of E_1. Further assume that we can determine an upper bound R_{max} on the possible reward for any state. Then, if in one of the other branches after S steps a probability of less than a certain P is accumulated and an up to then expected reward of E_2 is obtained, the branch can be pruned if and only if $E_2 + P \cdot (R_{max} \cdot (H - S)) < E_1$, where H is the planning horizon. This is true, because then this branch cannot anymore reach an expected reward greater than the one already found for the earlier branch. This, like in α-β-pruning [39], prunes the tree and can save computational effort in a sound way.

In real-time decision making, it is often crucial to have a decision made within a certain time. For that purpose our optimization procedure could be changed to an any-time algorithm that instead of an horizon takes a time as argument up to which the algorithm is run. This would require to move away from the current depth-first traversal to a breadth-first traversal procedure or some other strategy.

Also some of the ideas appearing in related work which we briefly mentioned in Section 5.2.4 could be worth integrating. These concerned methods for automatic decomposition of problems into sub-problems for which options could be generated [3] and homomorphisms between semi-MDPs [32] to tackle the problem of relatedness of sub-problems for which the same options could be applied.

[1] Imagine, for example, a gaussian distribution used to describe the actual position change of a robot after moving a certain distance.

Appendix A

ReadyLog Interpreter

See file “readylog.pl” on the CD.

A.1 definitions.pl

See file “definitions.pl” on the CD.

A.2 Transition Semantics

See file “final_trans.pl” on the CD.

A.3 Decision-theoretic Planning

See file “decisionTheoretic.pl” on the CD.

A.4 Options

See file “options.pl” on the CD.

A.5 Preprocessor

See file “preprocessor.pl” on the CD.

Bibliography

[1] R. A.Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

[2] A.McGovern, D.Precup, B.Ravindran, S.Singh, and R.S.Sutton. Hierarchical optimal control of mdps. In *Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems*, 1998.

[3] Eyal Amir and Barbara Engelhardt. Factored planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence 2003*. Computer Science Division, University of California at Berkeley, 2003.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, Princton, NJ, 1957.

[5] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the 17th National Conference on AI, AAAI 2000*, 2000.

[6] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[7] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order mdps. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence 2001*, 2001.

[8] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the AAAI 15th National Conference on Artificial Intelligence*, 1998.

[9] Carnegie Mellon University Robot Soccer Page. *http://www-2.cs.cmu.edu/˜robosoccer/simulator*.

[10] A M Cheadle, W Harvey, A J Sadler, J Schimpf, K Shen, and M G Wallace. Eclipse: An introduction. Technical report, IC-Parc, Imperial College London, 2003. Technical Report IC-Parc-03-1.

[11] Remco de Boer and Jelle Kok. The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. Master's thesis, University of Amsterdam, 2002.

[12] G. De Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. Technical report, Department of Computer Science, University of Toronto, Toronto, Canada, 1998.

[13] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[14] F. Dylla, A. Ferrein, and G. Lakemeyer. Acting and deliberating using golog in robotic soccer - a hybrid architecture. In *Third International Workshop on Cognitive Robotics*, 2002.

[15] Lin Fangzhen and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[16] A. Ferrein, C. Fritz, and G. Lakemeyer. Extending DTGolog with options. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence 2003*, 2003.

[17] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, 1998.

[18] Henrik Grosskreutz. *Towards More Realistic Logic-Based Robot Controllers*. PhD thesis, Aachen University, Germany, 2001.

[19] Henrik Grosskreutz and Gerhard Lakemeyer. cc-golog: Towards more realistic logic-based robot controllers. In *AAAI/IAAI*, pages 476–482, 2000.

[20] Henrik Grosskreutz and Gerhard Lakemeyer. Turning High-Level plans into robot programs in uncertain domains. In *ECAI*, pages 548–552, 2000.

[21] M. Hauskrecht, N. Meuleau, L. Kaelbling, T. Dean, and C. Boutilier. Hierarchical solutions of MDPs using macro-actions. In *Proc. UAI 98*, 1998.

[22] Norman Jansen. A framework for deliberation in uncertain, highly dynamic environments with real-time requirements. Masters Thesis, in German, Knowledge Based Systems Group, Aachen University, Aachen, Germany, 2002.

[23] Gerhard Lakemeyer. On sensing and off-line interpreting in Golog. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter*, pages 173–187. Springer, Berlin, 1999.

[24] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linkoping Electronic Articles in Computer and Information Science Vol. 3(1998): nr 018. URL: http://www.ep.liu.se/ea/cis/1998/018/.*, 1998.

[25] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[26] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[27] M.Chen, E.Foroughi, F.Heintz, Z.Huang, S.Kapetanaski, K.Kostiadis, J.Kummeneje, I.Noda, O.Obst, P.Riley, T.Steffens, Y.Wang, and X.Yin. Robocup soccer server. Technical report, 2001.

[28] Javier Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1994.

[29] Doina Precup and Richard S. Sutton. Multi-time models for temporally abstract planning. In M. Mozer, M. Jordan, and T. Petsche, editors, *NIPS-11*. MIT Press, 1998.

[30] Doina Precup, Richard S. Sutton, and Satinder P. Singh. Theoretical results on reinforcement learning with temporally abstract options. In *European Conference on Machine Learning*, pages 382–393, 1998.

[31] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, New York, 1994.

[32] Balaraman Ravindran and Andrew G. Barto. SMDP homomorphisms: An algebraic approach to abstraction in semi-markov decision processes. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence 2003*. Department of Computer Science, University of Massachusetts, 2003.

[33] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artifical Intelligence and Mathematic Theory of Comupation: Papers in Honor of John McCarthy*, 1991.

[34] R. Reiter. *Knowledge in Action*. MIT Press, 2001.

[35] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, second edition, 2003.

[36] Chair for Technical Computer Science, RWTH Aachen. *http://www.techinfo.rwth-aachen.de*.

[37] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence 2001*, pages 19–26, 2001.

[38] R. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Journal of Artificial Intelligence*, 1999.

[39] T.P.Hart and D.J.Edwards. The tree prune (tp) algorithm. Technical report, Massachusetts Institute of Technology, 1961. Artificial intelligence project memo 30.

[40] UvA Trilearn 2002. *http://carol.wins.uva.nl/˜jellekok/robocup/2002/index_en.html*.

[41] J. Wunderlich and F. Dylla. Technical specifications of the allemaniacs soccerrobots. Technical report, LTI / KBSG, Aachen University, Germany, 2002. in German.