# Logic-based Robot Control in Highly Dynamic Domains

Alexander Ferrein and Gerhard Lakemeyer

*Knowledge-Based Systems Group, RWTH Aachen University, Ahornstrasse 55, D-52056 Aachen, Germany*

**Abstract**

In this paper we present the robot programming and planning language Readylog, a Golog dialect which was developed to support the decision making of robots acting in dynamic real-time domains like robotic soccer. The formal framework of Readylog, which is based on the situation calculus, features imperative control structures like loops and procedures, allows for decision-theoretic planning, and accounts for a continuously changing world. We developed high-level controllers in Readylog for our soccer robots in RoboCup's Middle-size league, but also for service robots and for autonomous agents in interactive computer games. For a successful deployment of Readylog on a real robot it is also important to account for the control problem as a whole, integrating the low-level control of the robot (such as localization, navigation, and object recognition) with the logic-based high-level control. In doing so our approach can be seen as a step towards bridging the gap between the fields of robotics and knowledge representation.

*Key words:* reasoning about actions, planning, cognitive robotics, RoboCup

## 1 Introduction

Research on autonomous mobile robots has developed highly successful methods for tasks like localization, mapping, or navigation. In general, these methods rely heavily on techniques such as probabilistic state estimation, and there seems to be little if any need for ideas from logic-based knowledge representation (KR). On the other hand, the planning and reasoning-about-action communities in KR have made significant advances in modeling and reasoning

*Email addresses:* ferrein@cs.rwth-aachen.de, gerhard@cs.rwth-aachen.de (Alexander Ferrein and Gerhard Lakemeyer).

about the dynamics of the world. While there is perhaps little question that reasoning is useful when a robot needs to make decisions on how to achieve its goals, few KR techniques have actually found their way into robotic systems. One of the reasons is that these techniques are perceived as being far too computationally demanding. In this paper we want to demonstrate that this need not be so and that it is possible and indeed very fruitful to integrate state-of-the-art logical reasoning mechanisms into a robot. As we will see, this is even possible in domains like robotic soccer, where tight real-time constraints need to be taken into account. Enabling a robot to represent part of its environment using logic also forces one to take seriously the idea that symbolic representations need to be semantically grounded in how the robot's sensors perceive the world, an aspect usually ignored in the KR community. It is in this sense that we feel that the paper fits into the theme of *semantic knowledge* in robotics. Another aspect dealt with in the paper is that one cannot ignore that the world changes constantly, even when the robot is thinking about what to do next.

Here we present the robot programming and planning language Readylog, a derivative of Golog (Levesque et al., 1997), which is based on the situation calculus (McCarthy, 1963), a first-order logic to reason about actions and change. Readylog was designed to support high-level decision making for robots acting in dynamic real-time domains like robotic soccer. The idea of Readylog is to combine planning with programming. It features control structures known from imperative programming languages, but also non-standard constructs for decision-theoretic planning employing MDP theories in the logical framework. Readylog supports reasoning under uncertainty and continuous change. It accounts for problems like gathering sensor values and integrating them into the high-level controller, or monitoring the execution of previously established behavior plans. It was successfully applied for controlling (soccer) robots in RoboCup competitions (Ferrein et al., 2005; Ferrein, 2008) and service robotics tasks (Schiffer et al., 2006a), and it showed its usefulness as a behavior representation language (Dylla et al., 2008). With the embedding of the Readylog high-level controller on a real robot and integrating it into the overall robotic system, we demonstrate an approach to bridge the gap between robotics and logic-based methods from the field of reasoning about actions. We want to emphasize that, for our work, the combination of the low-level control of the robot (localization, navigation, object recognition) with the logic-based high-level control was of great importance to come up with an overall efficient and flexible robot controller.

The paper is organized as follows. In the next section we discuss related work. In Section 3, we show the robotics side of this work. We describe the hardware platform of our robots and their software system. In Section 4 we present some of the theoretical background of the situation calculus and Golog, which Readylog is based on. Section 5 addresses our approach to the control

problem for robots acting in dynamic domains. Readylog makes extensive use of decision-theoretic (DT) planning, and we show how DT planning works in the Readylog framework. We further address the problem of monitoring the execution of the behavior policies, and the problem of integrating sensor values in an efficient way. At the end of this section, we present an approach to macro actions in the decision-theoretic context to speed up planning. In our examples we show how Readylog is used for controlling soccer robots in RoboCup's Middle-size league. Other Readylog applications are discussed in Section 6. Then we conclude.

## 2 Related Work

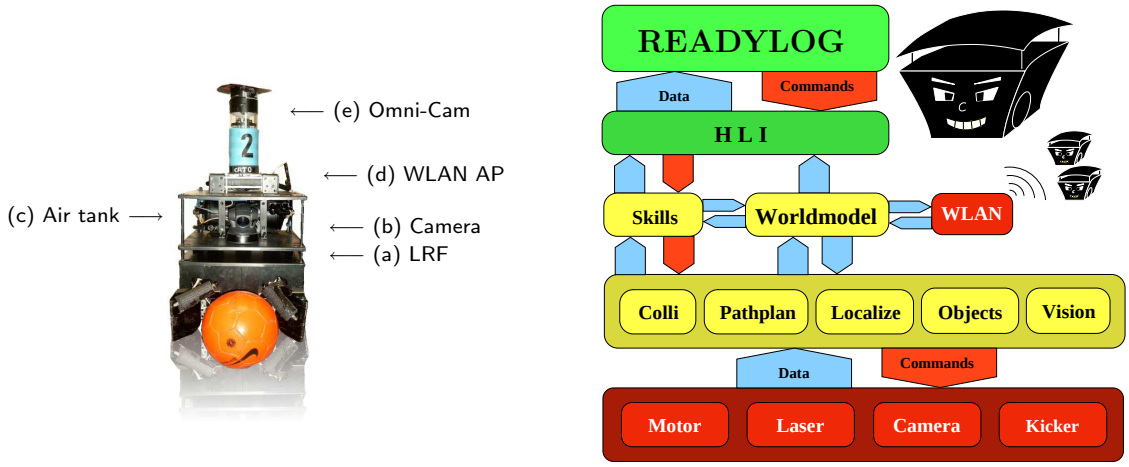
For the problem of high-level decision making a rich body of related approaches exist. These comprise work on decision making in general, but also on applications from various fields. Here, we restrict ourselves to research on applying decision making techniques to mobile robots. Some examples are the *Procedural Reasoning System* (PRS) (Ingrand et al., 1996), PRS-lite (Myers, 1996), the Saphira architecture (Konolige et al., 1997), *Reactive Action Packages* (RAP) (Bonasso et al., 1997), or the *Reactive Plan Language* (RPL) (McDermott, 1991) and *Structured Reactive Controller* (SRC) (Beetz, 2001). Although these approaches follow different directions, they have often influenced each other. For example, language constructs which express that a process waits for certain conditions to become true have been incorporated in many languages, including RPL and also Readylog. There are other approaches that use Markov Decision Processes (MDPs) for decision making of a robot. One impressing example for deploying this techniques is Nursebot (Pineau et al., 2003). The robot was applied in nursing homes to help the elderly with their daily life, reminding to take medicine or to go the doctor's.

Another interesting and related field for robot control is the high-level control of autonomous rovers which fulfill planetary missions. The goals of a mission planner are different from ours. The domain is generally less dynamic, but resource allocation and run-time plan adaptation need to be taken into account. Further, the terrain the robot is operating in is rough and unknown. For example, Lemai and Ingrand (2004) report on a partial order planner which also accounts for plan repair. In (Finzi et al., 2004) the authors describe a rover deploying a model-based planning approach based on the Intelligent Distributed Execution Architecture (Muscettola et al., 2004). A similar approach is followed by Carbone et al. (2008) where a model-based high-level controller for rescue robots is presented. They also concentrate on integrating sensor values into the model-based controller of the robot. They tested their architecture on real robots in search & rescue scenarios.

On the side of logic-based action formalisms a closely related approach is the work of Pirri et al. (2003), where an architecture for a domestic robot is presented. They also make use of GOLOG for their high-level control, and stress the importance of representing the knowledge about the environment and of integrating sensing results into the high-level controller as well as of monitoring the execution of the robot's action. However, they do not deal with domains where real-time decision making is as important as in our case. The first realistic large-scale application of GOLOG was the tour-guide robot in the Deutsches Museum Bonn in the Rhino project (Burgard et al., 1998). Over several days a RWI B21 robot served as museum tour-guide and explained the exhibits. Funge (2000) makes use of Golog for modeling animated creatures in a cognitive way. He uses the possibility of non-determinism in Golog for his creatures to fill in details in sketch plans based on their background domain knowledge. Levesque and Pagnucco (2000) report on Legolog, their implementation of GOLOG on a Lego Mindstorm robot. They connected an IndiGolog interpreter implemented in Prolog to the Lego Mindstorm Robotics Invention System (RIS). Pham (2006) describes an interface between DTGolog and the Sony Aibo ERS-7. The interface is based on the framework Tekkotsu (Tira-Thompson, 2004). For example, an application that the Aibo is used for is to fulfill navigation tasks for which an optimal policy was calculated.

The Fluent calculus (Thielscher, 1998) is an approach to reasoning about actions and change similar to the situation calculus. With FLUX (Thielscher, 2005), which stands for FLUent eXecutor, Thielscher introduces a run-time system for the fluent calculus. FLUX was applied to mobile service robots (Thielscher, 2000), but also showed its strength at the 2006 AAAI General Game Playing Project Competition (Genesereth et al., 2005) by winning the competition (Schiffel and Thielscher, 2007). Other approaches for reasoning about actions are (Pednault, 1989; Gelfond and Lifschitz, 1993; Doherty et al., 1998; Sandewall, 1998), to name but a few. Sandewall (1998) proposes the *Cognitive Robotics Logic* (CRL). He presents a meta-theory for reasoning about actions. The language allows for expressing durative actions, composite actions, nondeterministic actions, nondeterministic timing of actions and their effects, continuous time and piecewise continuous fluents, imprecise sensors and actuators, and action failures. Similar to CRL, the temporal action logic TAL (Doherty et al., 1998; Kvarnström et al., 2000) makes use of a surface language representing narratives, and a base language allowing the agent to reason about narratives. The language TAL is also applied to deliberative tasks for unmanned aerial vehicles (Doherty, 2005).

Another recent paper underlining the importance of integrating semantic knowledge into the robot controller is (Bouguerra et al., 2007), where description logics was chosen to represent the knowledge the robot has about its environment. They propose a monitoring scheme where sensor values are verified against the semantic background knowledge. An early approach which




(a) AllemaniACs Robot (b) Software Architecture

Fig. 1. The *AllemaniACs* System.

makes use of description logics for controlling robots is (Giacomo et al., 1997). The authors describe an approach to high-level control similar to the situation calculus. Many other papers concentrate on integrating knowledge into the robot system or use logic-based approaches to control a robot and to integrate semantic knowledge of the environment into the controller. For further reference on logic-based approaches to robot control, we refer to (Levesque and Lakemeyer, 2007). In this recent paper, a concise overview of the current developments and streams in the field of *reasoning about actions* and the *Cognitive Robotics* community is given.

## 3 Robot Platform and Software Architecture

As we pointed out in the introduction, there are three parts involved in our work, one of which is the robot system with its sensors and actuators and its low-level control software. In this section we briefly go over our robot system.

### *3.1 The Robot Platform*

The hardware platform of our Middle-size RoboCup Team *AllemaniACs* has a size of 39 cm × 39 cm × 80 cm (Figure 1(a)) with a weight of 70 kg. The robot has a differential drive, each motor has 2.4 kW power. The motors were originally developed for electric wheel chairs. They provide the robot with a translational top speed of about 2.5 to 3 m/sec. The two 12 V lead-gel batteries with 15 Ah allow for 2 hours operation time. Fig. 1(a) shows the hardware platform. Directly above the base a 360° laser range finder (LRF) is mounted

which provides data at 10 Hz. A Sony EVI-D100P camera (marked as (b) in Fig. 1(a)) is installed yielding images in PAL resolution with 25 Hz. Behind the camera, parts of the air tank for our pneumatic kicking device becomes visible (Item (c) in Fig. 1(a)). On top, the IEEE 802.11a/b/g access point for wireless communication and an omni-directional camera, pointing to a hyperbolic mirror, is mounted ((d) and (e) in the figure). The robot has two on-board Pentium III PCs at 933 MHz running Linux, one equipped with a frame-grabber for the Sony EVI-D100P camera. This platform allows for soccer playing, but is also used for service robotics applications as in *RoboCup@Home* (Schiffer et al., 2006a).

### *3.2 Robot Control Software*

In this section we take a closer look at the low-level control software of the AllemaniACs robots. The system uses a classical three layered architecture with an interface layer between the hardware and the control modules on the middle layer, which in turn builds the interface to our high-level decision making with Readylog. The middle layer features modules like navigation, localization, or object recognition. The third layer of the system architecture consists of the world model and the reasoning component Readylog, which we will introduce in Section 5. The software architecture is shown in Figure 1(b). The control flow is, as is usual in layered hierarchical architectures, from bottom to top concerning data, and from top to bottom w.r.t. control commands (cf. e.g. (Murphy, 2000)). For communication between control modules we make use of a blackboard system. Each module connects to the blackboard system and is able to read data provided by other modules from the blackboard. Inside the blackboard several data sections are separated and access rights are regulated.

The low-level interfaces are basically hardware drivers with access to the blackboard. The *motor* driver provides data like odometry information which are calculated from the wheel encoders and estimates about the velocity of the robot on the one hand, on the other hand it takes driving commands from modules of upper levels. The *laser* driver takes commands for starting or stopping the LRF, and provides 360 distance measurements per sweep. As the *directed camera* is with its pan/tilt unit also an actuator it can take commands like $move(\varphi, \theta)$. It provides the vision module with camera images. The omnivision camera only yields raw images. Finally, we have the *kicker* interface which takes commands actuating the pressure valves of the pneumatic kicking device.

The modules on the middle layer work on the data provided by the sensors. A central task especially with fast heavy-weight robots is an effective *colli-*

*sion avoidance* strategy. With the data from the LRF we create an occupancy grid map, and then search for a collision-free path in it. For successful complex robot operations in dynamic environments, moreover, good localization is needed. Here we make use of a Monte Carlo approach (Strack et al., 2006). To endow the robot's world model with a rich representation of the environment, one further needs good object classification. We use the information provided by the vision module, and further we make use of the fact that the robot is localized in its given environment map. Thus, it is able to distinguish between static and dynamic obstacles. The dynamic obstacles are classified by their laser signature. For the soccer application, an important feature is the ball. The *vision* module inspects a camera image on several scan lines of a color segmented image. For finding the ball we apply randomized circle fitting. The circle fitting is implemented as an any-time algorithm which returns the best fitted circle. With a geometric model of the robot the position of the ball is estimated.

Above the middle layer, Figure 1(b) shows the modules *world model*, and *skills*. These are the modules with which our high-level framework READYLOG is connected. From the point of view of high-level decision making, the skill module encapsulates actuators, the world model encapsulates sensor data. The skill module provides the basic actions for READYLOG. These are for example actions like *drive to global position* or *turn with angle* $\theta$ or more sophisticated ones like *dribble around opponents*. While the basic actions are clearly influenced by the soccer application, they are nonetheless useful for service robotics applications as well. As our high-level controller, which we present in detail in Section 5 is implemented in Prolog and the rest of the software is implemented in C++, we need another interface between READYLOG and the low-level control software. This function has the module *HLI*, the high-level interface. It translates Prolog calls to appropriate C++ function calls.

## 4 The Situation Calculus and Golog

### *4.1 Situation Calculus*

The situation calculus is a second order language with equality which allows for reasoning about actions and their effects. The world evolves from an initial situation due to primitive actions. Possible world histories are represented by sequences of actions. The situation calculus distinguishes three sorts: *actions*, *situations*, and domain dependent *objects*. A special binary function symbol $do : action \times situation \rightarrow situation$ exists, with $do(a, s)$ denoting the situation which arises after performing action $a$ in the situation $s$. The constant $S_0$ denotes the initial situation, i.e. the situation where no actions have yet

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

For each action one has to specify a *precondition axiom* stating under which conditions it is possible to perform the respective action and an *effect axiom* formulating how the action changes the world in terms of the specified fluents. An action precondition axiom has the form $Poss(a(\vec{x}), s) \equiv \Phi(\vec{x}, s)$ where the binary predicate $Poss \subseteq action \times situation$ specifies when an action can be executed, and $\vec{x}$ stands for the arguments of action $a$. For our travel action the precondition axiom may be $Poss(goto(room), s) \equiv robotLoc(s) \neq room$. After having specified when it is physically possible to perform an action it remains to state how the respective action changes the world. In the situation calculus the effects of actions are formalized by so-called successor state axioms of the form $F(\vec{x}, do(a, s)) \equiv \varphi_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\varphi_F^-(\vec{x}, a, s)$, where $F$ denotes a fluent, $\varphi_F^+$ and $\varphi_F^-$ are formulas describing under which conditions $F$ is true, or false resp. This axiom simply states that $F$ is true after performing action $a$ if $\varphi_F^+$ holds, or the fluent keeps its former value if it was not made false. Successor state axioms describe Reiter's solution to the frame problem (Reiter, 2001), the problem that all the non-effects of an action have to be formalized as well. As an example, consider the following successor state axiom for the fluent $robotLoc(s) : robotLoc(do(a, s)) = y \equiv a = goto(room) \wedge y = room \vee a \neq goto(room) \wedge y = robotLoc(s)$. Note that free variables in the occurring formulas are meant to be implicitly universally quantified. The background theory is a set of sentences $\mathcal{D}$ consisting of $\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$, where $\mathcal{D}_{ssa}$ contains sentences about the successor state axioms, $\mathcal{D}_{ap}$ contains the action precondition axioms, $\mathcal{D}_{una}$ states sentences about unique names for actions, and $\mathcal{D}_{S_0}$ consists of axioms what holds in the initial situation. Additionally, $\Sigma$ contains a number of foundational axioms defining situations. For details we refer to (Pirri and Reiter, 1999; Reiter, 2001).

### *4.2 Golog*

The high-level programming language Golog (Levesque et al., 1997) is based on the situation calculus. As planning is known to be computationally very demanding in general, which makes it impractical for deriving complex behaviors with hundreds of actions, Golog finds a compromise between planning

and programming. The robot or agent is equipped with a situation calculus background theory. The programmer can specify the behavior just like in ordinary imperative programming languages but also has the possibility to project actions into the future. The amount of planning (projection) used is in the hand of the programmer. With this, one has a powerful language for specifying the behaviors of a cognitive robot or agent. While the original GOLOG is well-suited to reason about actions and their effects, it has the drawback that a program has to be evaluated up to the end before the first action can be performed. It might be that the world changed between plan generation and plan execution so that the plan is not appropriate or is invalid. To overcome this problem, De Giacomo et al. (2000) proposed an incremental interpreter with CONGOLOG. The program is interpreted in a step-by-step fashion where a transition relation defines the transformation from one step to another. In this so-called transition semantics a program is interpreted from one configuration $\langle \sigma, s \rangle$, a program $\sigma$ in a situation $s$, to another configuration $\langle \delta, s' \rangle$ which results after executing the first action of $\sigma$, where $\delta$ is the remaining program and $s'$ the situation resulting of the execution of the first action of $\sigma$. The one-step transition function *Trans* defines the successor configuration for each program construct. In addition, another predicate $Final$ is needed to characterize final configurations, which are those where a program is allowed to legally terminate.

To illustrate the transition semantics, let us consider the definition of *Trans* for some of the language constructs:

(1) $Trans(nil, s, \delta, s') \equiv false$
(2) $Trans(\alpha, s, \delta, s') \equiv Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$
(3) $Trans([\sigma_1; \sigma_2], s, \delta, s') \equiv Final(\sigma_1, s) \wedge$
$Trans(\sigma_2, s, \delta, s') \vee \exists \delta'.\delta = (\delta'; \sigma_2) \wedge Trans(\sigma_1, s, \delta', s)$
(4) $Trans(\sigma_1 \,||\, \sigma_2, s, \delta, s') \equiv$
$\exists \gamma.\delta = (\gamma \,||\, \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee \exists \gamma.\delta(\sigma_1 \,||\, \gamma) \wedge Trans(\sigma_2, s, \gamma, s')$

(1) Here *nil* is the empty program, which does not admit any further transitions.
(2) For a *primitive action* $\alpha$ we first test if its precondition holds. The successor configuration is $\langle nil, do(\alpha, s) \rangle$, that is, executing $\alpha$ leads to a new situation $do(\alpha, s)$ with the *nil* program remaining.
(3) The next definition concerns an action sequence $[\delta_1; \delta_2]$, where it is checked whether the first program is already final and a transition exists for the second program $\delta_2$, otherwise a transition of $\delta_1$ is taken.
(4) $\sigma_1 \,||\, \sigma_2$ denotes that $\sigma_1$ and $\sigma_2$ can be executed concurrently. Here the definition of $Trans$ makes sure that one of the two programs is allowed to make a transition without specifying which. This corresponds to the usual interleaved semantics of concurrency.

We only sketched the transition semantics here. In the next section, some more examples are given. For a concise overview of the transition semantics we refer the interested reader for example to (De Giacomo et al., 2000, 2001). We remark that the transition semantics allows for a natural integration of sensing and on-line execution of programs.

## 5 Readylog – Real-time and Dynamic Golog

### *5.1 Introduction*

The aim of designing the language READYLOG was to create a GOLOG dialect which supports the programming of the high-level control of agents or robots in dynamic real-time domains. Our primary application was robotic soccer. The robotic soccer domain has some specific characteristic which made the development of READYLOG necessary and influenced several design decisions: the robotic soccer domain is an unpredictable adversarial dynamic real-time domain. This means that decisions have to be taken quickly and making plans for future courses of actions have a mid-term horizon. Planning ahead for the next minute does not make sense as the world changes unpredictably due to the uncertainty of the outcomes of the own actions and the behaviors of the opponent players. The unpredictability of the actions of the agent demands for some notion of uncertainty. The idea of GOLOG to combine planning with programming was accounted for by integrating decision-theoretic planning; only partially specified programs which leave certain decisions open, which then are taken by the controller based on an optimization theory, are needed. READYLOG borrows ideas from (Levesque et al., 1997; Grosskreutz and Lakemeyer, 2001; De Giacomo et al., 2000; Grosskreutz, 2000; Boutilier et al., 2000) and features the constructs given in Fig. 2. We will not introduce the whole semantics of READYLOG, here. Some of the constructs have already been introduced in the previous section. Other constructs like the *solve* statement will be discussed in detail in the next section. While the aforementioned extensions were integrated into one framework, this was not enough to deploy GOLOG in dynamic real-time domains as robotic soccer. In particular, one needs an efficient implementation accounting for the real-time constraints posed by the environment. Therefore, the READYLOG framework features

(1) a novel on-line version of the decision-theoretic planning method proposed by Boutilier et al. (2000), which allows for execution monitoring of policies;
(2) an enhanced version of passive sensing which allows for updating the world model in the background;
(3) the introduction of macro actions, so-called options, for decision-theoretic

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

Fig. 2. Overview of Readylog constructs

planning, based on Precup et al. (1998);

(4) several speed-ups for policy generation such as making use of caching previously computed results in the forward decision-theoretic search for an optimal policy;

(5) a useful any-time approach for decision-theoretic planning to overcome fixed horizons when searching for a policy and by this to better exploit the computational resources of the agent or robot, and

(6) a progression method based on Lin and Reiter (1997).

We address issues 1–3 in the following. For reasons of space we cannot discuss the issues 4, 5 and 6 here. For more details about these topics we refer to (Ferrein, 2008).

### *5.2 DT Planning in Readylog*

One of the most important features to model the behavior of our robots is the use of decision-theoretic planning. It is very convenient, as the domain axiomatizer may leave open several choices in her behavior specification. The Readylog interpreter chooses the best action alternative based on the un-

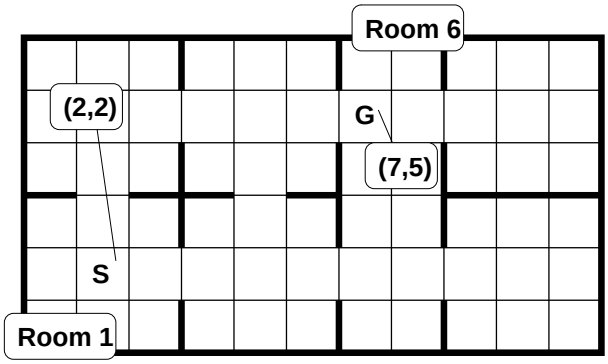



Fig. 3. The Maze66 domain from Hauskrecht et al. (1998).

derlying utility theory. To illustrate how READYLOG calculates an optimal policy from a given input program we give a navigation example from a toy maze domain. A robot wants to navigate from its start position $S$ to a goal position $G$. It can perform one of the actions from the set $A = \{go\_right, go\_left, go\_up, go\_down\}$. Each of the actions brings the robot to one of its neighboring locations. The actions are stochastic, that is there exists a probability distribution over the effects of the action. Each action takes the agent to the intended field with probability of $p$, with probability $1 - p$ the robot will arrive at any other adjacent field. The maze shown in Fig. 3 is the well-known Maze66 domain from (Hauskrecht et al., 1998). The robot cannot go through the walls, if it tries, though, the effect is that it does not change its position at all.

The fluent *goal* defines the goal position ($goal = (7, 5)$), and the fluent *loc* denotes the current position of the robot in the maze. The reward function is defined as $reward(s) = +1$ if $loc(s) = goal$ and $-1$ otherwise. To find the optimal path from $S$ to $G$ the robot is equipped with the program

**proc** *navigate*
  **solve**($h$, *reward*, **while** $loc \neq goal$ **do**
    ($go\_right \mid go\_left \mid go\_up \mid go\_down$)
  **endwhile**, $h$)
**endproc**

With the *solve* statement decision-theoretic planning is initiated. The interpreter switches into an off-line mode and optimizes the program given as the argument of the solve-statement up to horizon (number of actions) $h$. The "|" represent a nondeterministic action choice. At these choice points the interpreter selects the best action alternative. The READYLOG interpreter does this via predicates *BestDo* which implement the forward-search algorithm (Fig. 4). For space reasons we will not show the whole definition of the algorithm here. For a detailed discussion of *BestDo* we refer to (Boutilier et al., 2000; Ferrein, 2008). As long as the robot is not at the goal location (and the

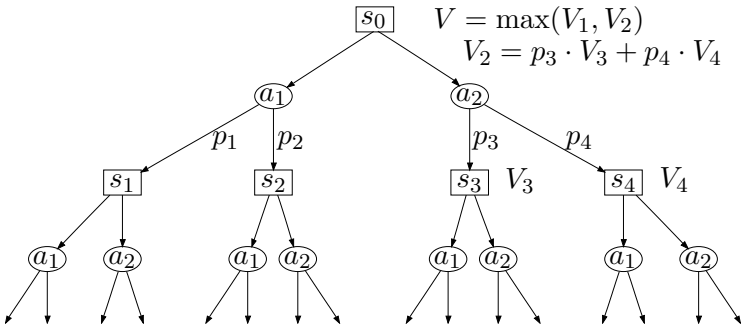



Fig. 4. Decision tree search in READYLOG

horizon is not reached), READYLOG loops over the nondeterministic choice statement. At each iteration the interpreter expands a sub-tree for each of the actions inside the choice statement. As each of the actions are stochastic, again for each outcome of each action the interpreter branches over all possible nature's choices. This process iterates until either the agent is located at the goal position or the horizon is reached. At the leaves of the computation tree over *BestDo* (at the end of the recursion) the agent receives the reward for these final situations. Then, "going up" the computation tree for nondeterministic choices, the best alternative is evaluated and chosen for the policy. An illustration of the computation tree is given in Fig. 4. Since it is not known in advance which outcome is chosen by nature at execution time, the policy needs to cover all possibilities, which is realized by nested conditionals. Coming up to the root node, the computation terminates returning the policy, the value for the policy, and its probability of success.

### *5.3 Execution Monitoring for Policies*

As we remarked in the introduction to this section, our specification language should not only be able to calculate a policy, but should also be able to execute the previously established policy. Therefore, we need a run-time environment for executing policies on-line. Soutchanski (2001) proposed an on-line variant of DTGOLOG, the decision-theoretic variant of GOLOG. In an earlier paper we showed that his approach is in general not feasible in real-time domains (see (Ferrein et al., 2004; Ferrein, 2008) for a detailed discussion on that matter). In our approach, we introduce the operator $solve(h, f, p)$ for a program $p$, a reward function $f$, and a fixed horizon $h$, which initiates decision-theoretic planning in the on-line transition semantics.

$$Trans(\mathbf{solve}(h, f, p), s, \delta, s') \equiv$$
$$\exists \pi, v, pr.BestDo(p, s, h, \pi, v, pr, f) \wedge \ \delta = applyPol(\pi) \wedge s' = s$$

The predicate *BestDo* first calculates the policy for the whole program $p$. The policy $\pi$ is then scheduled for on-line execution as the remaining program. We remark that policy generation assumes that the program does not contain explicit sensing actions. As we will see in the next subsection this accounts for so-called passive sensing. In the case of the robot's position, for example, policy generation works with an abstract model of the robot's movements so that robot positions in future states can simply be computed without having to appeal to actual sensing. Making use of such models during plan generation requires that we monitor whether $\pi$ remains valid during execution as discrepancies between the model and the real-word situation might arise. Monitoring is handled within *applyPol*, which is defined below. Note that the *solve* statement never reaches a final configuration as further transitions are needed to execute the calculated policy. To keep track of the model assumptions we made during planning, we introduce special markers into the policy. Hence, in the definition of *BestDo* we have to store the truth values of logical formulas. For conditionals this means:

$$
\begin{array}{l}
\mathit{BestDo}(\textbf{if } \varphi \textbf{ then } p_1 \textbf{ else } p_2 \textbf{ endif}; p, s, h, \pi, v, pr) \doteq \\
\varphi[s] \wedge \exists \pi_1 . \mathit{BestDo}(p_1; p, s, h, \pi_1, v, pr) \wedge \pi = \mathfrak{M}(\varphi, \mathit{true}); \pi_1 \vee \\
\neg\varphi[s] \wedge \exists \pi_2 . \mathit{BestDo}(p_2; p, s, h, \pi_2, v, pr) \wedge \pi = \mathfrak{M}(\varphi, \mathit{false}); \pi_2
\end{array}
$$

Thus, for conditionals we introduce a marker into the policy that keeps track of the truth value of the loop condition at planning time. We prefix the generated policy with a marker $\mathfrak{M}(\varphi, \mathit{true})$ in case $\varphi$ turned out to be true in $s$ and $\mathfrak{M}(\varphi, \mathit{false})$ otherwise. While-loops are treated in a similar way. The treatment of a test action $\varphi?$ is even simpler, since only the case where $\varphi$ is true matters. If $\varphi$ is false, the current branch of the policy is terminated, which is indicated by the *Stop* action.

$$
\begin{array}{l}
\mathit{BestDo}(\varphi?; p, s, h, \pi, v, pr) \doteq \\
\varphi[s] \wedge \exists \pi' . \mathit{BestDo}(p, s, h, \pi', v, pr) \wedge \pi = \mathfrak{M}(\varphi, \mathit{true}); \pi' \vee \\
\neg\varphi[s] \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = \mathit{reward}(s)
\end{array}
$$

Next, we will show how our annotations will allow us to check at execution time whether the truth value of conditions in the program at planning time are still the same and what to do about it when they are not. In case a marker was inserted into the policy we have to check whether the test performed at planning time still yields the same result. If this is the case we are happy and continue executing the policy, that is, *applyPol* remains in effect in the successor configuration. But what should we do if the test turns out different? We have chosen to simply abort the policy in our current formalization, that

is, the successor configuration has $Nil$ as its program. Expressed formally, this means:

$$\begin{aligned}&Trans(applyPol(\mathfrak{M}(\varphi, v); \pi), s, \delta, s') \equiv s = s' \wedge \\ &(v = true \wedge \varphi[s] \wedge \delta = applyPol(\pi) \vee v = true \wedge \neg\varphi[s] \wedge \delta = Nil \vee \\ &v = false \wedge \neg\varphi[s] \wedge \delta = applyPol(\pi) \vee v = false \wedge \varphi[s] \wedge \delta = Nil)\end{aligned}$$

Note that in the above formula $\varphi[s]$ stands for the logical formula where previously suppressed situation arguments in fluents are restored. The *applyPol* transition further has to be defined for primitive and stochastic action as well as for conditionals. Due to changes in the world it may be the case that action $a$ has become impossible to execute. In this case we again abort the rest of the policy with the successor configuration $\langle Nil, s\rangle$. For an if-construct, which was inserted into the policy due to a stochastic action, we determine which branch of the policy to choose and go on with the execution of that branch. If we reach the horizon we have to stop the execution of the policy, which, if nothing went wrong, has reached a final configuration by then, i.e. $Final(applyPol(p, h), s) \equiv Final(p, s) \vee h = 0$. With the above definitions we are able to detect when a policy becomes invalid during execution. As stated above, currently, we handle invalid policies by simply invoking re-planning. For the complete definition and thorough discussion of *applyPol* as well as of *BestDo* we refer again to (Ferrein, 2008).

### *5.4 On-line Passive Sensing*

To deal with incomplete knowledge about the environment Golog was extended with sensing actions (Lakemeyer, 1999; De Giacomo and Levesque, 1999). These special actions allow an agent to query its sensors to gather information about the environment. This approach has, under certain circumstances, several drawbacks. When sensor values must be updated very frequently, acquiring world information through explicit sensing actions is not feasible. The agent would simply be overwhelmed with executing sensing actions. Another problem exists when off-line planning is interleaved with on-line execution. If the plan relies on the on-line information the result of planning might be inconsistent due to wrong sensing results. So, in general, with an *active sensing* approach it is not possible to plan ahead of sensing actions. What is needed is a passive sensing approach which performs updates of the sensor values in the background. Proposals for passive sensing approaches can be found in (Poole, 1997; Grosskreutz and Lakemeyer, 2001). Note the difference between active and passive sensing: with a passive sensing approach, there is no need to explicitly query sensors in the control program. (Here sensors may refer to quite abstract notions like a robot's position.) When deliberating, the robot would use (probabilistic) models of sensor values, and during execution

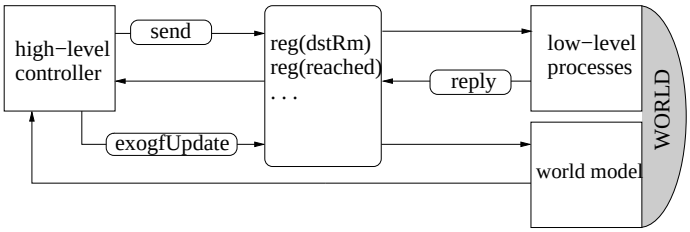



Fig. 5. The Extended Architecture

these are substituted by the actual values, usually supplied automatically at regular intervals. Active sensing, on the other hand, refers to explicit sensing actions which are part of a control program or plan and where reasoning usually involves a costly case analysis of the possible outcomes.

Updating a complete world model in simulated soccer domain, for example, takes more than 100 ms which is longer than the decision cycle in RoboCup's simulation league. To deal with this problem we extended the system architecture proposed in (Grosskreutz and Lakemeyer, 2001) with an explicit world model. They proposed a system architecture where the high-level controller starts low-level processes via a so-called action register. To initiate an action the high-level controller sends a command to the register. This command is passed through to the execution module which in turn cares for the execution of the action in the real world. From now on, the high-level controller is no longer concerned with monitoring the execution of the action. This is done asynchronously by the execution layer of the robotic system. When the execution of the action is finalized, the low-level control indicates this by sending a reply message to the high-level controller via the action register. The high-level controller can now react on this specific message. Between a *send* and a *reply* message the high-level controller could care for other things. The communication between high-level and low-level system is realized through a special fluent *register* and the two actions *send* and *reply*. The high-level controller invokes the low-level process with a *send* action, and the low-level process answers with a *reply* message when the low-level process is finished. In Figure 5 we present the extended system architecture. As in (Grosskreutz and Lakemeyer, 2001) we use the special *register* fluent and the message passing between high-level and low-level control. The presence of an explicit world model extends the architecture. The special action *exogfUpdate* which is sent via the register initiates an update of the world model of the agent. The shaded arc "world" denotes the connection to the real world. One should think of this in terms of the layered system architecture as shown in Section 3, which has access to the actuators and can gather data from the sensors. The possibility to asynchronously update sensor values allows for on-line passive sensing.

In this section, we present a READYLOG example from the soccer domain. We used the controller code shown below on our Middle-size robots at several competitions. As presented in Section 3, the *skill module* encapsulates the primitive actions including actions like *goto*, *intercept*, *dribble*, or *shoot*. The world model of the soccer robot comprises fluents like the agent position, the position of the ball, and the opponents. All this information comes together with confidences or visibility flags (which is true if the ball is seen), and come with two flavors: the agent could query the positions gathered locally from its sensors, or ask the information from the global world model. The information coming from the global world model are most of the times more accurate, though, they have some latency. The robot must also store tactical information, like its tactical role. Further, a predicate *bestInterceptor* calculates which player is best located to the ball and shall gain control over the ball. In the following we show an example from our attacking player from RoboCup's Middle-size league.

```
proc attackerBestInterceptor
    if scoringSituation then scoreDirectly(own)
    else if ¬haveBall then interceptBall(own, fast) endif
    endif
5     solve(4, reward,
        continueSkill(currentSkill); (haveBall)?; (kickTo(own)
        | dribbleAndKick(own)
        | dribbleToPoints(own)
        | if isKickable(own) then
10              pickBest(angle, {−3.1, −2.3, 2.3, 3.1}, /* in rad */
                 (turnRelative(own, angle, medium);
                 (intercepBall(own, slow); dribbleOrMoveKick(own)
                 | interceptBall(numberByRole(supporter)
                     dribbleOrMoveKick(numberByRole(supporter)
                   )/* end pickBest */
15          else
        interceptBall(own); dribbleOrMoveKick(own)
        | interceptBall(own, 0.0)
    endif)/* end solve */
20 endproc
```

In line 5, decision-theoretic planning is initiated using the reward function *reward*. The reward function, basically, gives a high reward for positions in front of the opponent goal, and high negative reward for situations in front of the own goal. The agent has the choice to calculate the best action among *kickTo* (l. 6), *dribbeMoveKick* (l. 7), the *dribbleToPoint* (l. 8), and a multi-

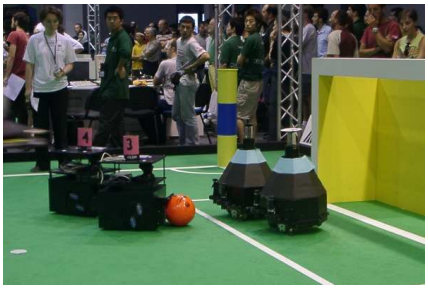
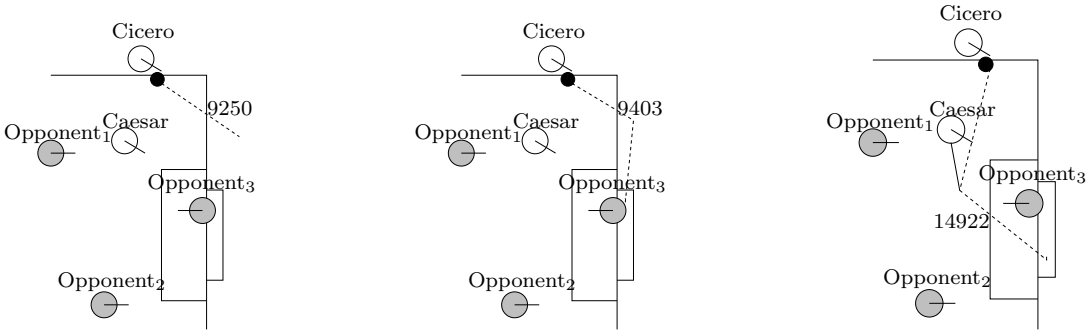


Fig. 6. A scene from the RoboCup 2004 against the Osaka team (right-hand side).




(a) Direct kick alternative.

(b) DribbleMoveKick alternative.

(c) Cooperative Play alternative.

Fig. 7. Game situations

agent plan (ll. 9–14) which we address below. The *kickTo* action directly kicks the ball, *dribbleMoveKick* action combines a dribbling with a goal shot if possible, *dribbleToPoints* stands for an action which lets the agent dribble to certain defined positions on the field. Fig. 7(a) shows the effect of the kick action (according to its action model), Fig. 7(b) shows the *dribbleMoveKick* model, while Fig. 7(c) shows the effect of the multi-agent plan from the lines 9–14 in the program above. With the *pickBest* statement several angles are chosen which serves as an argument for the following turn action. With this turn action, the robot plays a pass to its teammate, which in turn plans to intercept the ball and try a goal shot.

### *5.6 A DT Plan Library for Abstracted Plans*

Empirical results from several RoboCup competitions showed that decision-theoretic planning using the above program took between 0.17 and 2.1 seconds, with an overall average of 0.5 seconds. While an average reasoning time of 0.5 seconds may be acceptable in robotic soccer, 2 seconds are probably not because of the high risk that an opponent will attempt a tackle and intercept the ball. One possibility to speed up the computation is to make use of macro actions in the DT context. The idea is to define a sub-task, calculate an

optimal policy for the sub-task, and use the solution like a primitive action when solving more complex tasks later on. We proposed the use of such macro actions, or options, in the READYLOG framework in (Ferrein et al., 2003) leading to an exponential speed-up in the computation time. For dynamic domains like robotic soccer, however, this approach has some drawbacks, which are mainly related to the fact that the state space of the MDP has to be enumerated explicitly. To overcome these problems, we propose to calculate policies in an abstract way and store these abstract policies in a DT plan library. Later, when the agent can apply a policy from the plan library, it has just to instantiate the abstract policy without the need to plan from scratch.

The basic procedure to calculate a policy for a macro action is the following:

(1) *Off-line pre-processing*
   (a) Calculate an abstract policy for each solve statement occurring in the behavior specification.
   (b) Replace each solve statement with its abstract policy in the specification.
(2) *On-line execution*
   (a) Look up the policy, value, and probability of success for the option in the plan library.
   (b) If the option is not contained in the library, instantiate the option in the particular situation and store the value and the probability of success together with the current world state in the library.

An abstract policy is calculated as follows. In a run of the forward-search value iteration algorithm (READYLOG's DT planning algorithm), we do not calculate explicit numeric values for the reward function but keep them as abstract terms. Basically, we store the whole computation tree for a respective input program without optimizing away the agent's choices. Later, when instantiating a plan from the plan library, we can establish the optimal policy, the values and probabilities of all outcomes of the policy by evaluating this abstract policy. As an example for an abstract value for a policy, consider the maze domain where the agent wants to leave the first room through the northern door. The location at the northern door will have a high value, as this is the goal of the agent. But instead of calculating the concrete value, the value function is kept as the term $v = +(-(reward(do(go\_up, s)), cost(do(go\_up, s))), \cdot(prob(go\_up, up, s), \ldots$. Later, we only have to fill in a concrete world situation to get the respective value for leaving the room through the northern door. Similarly, we keep the policy as an abstract term, as well as the probability of its success. In the source code of our robot control program, we then replace every occurrence of a solve statement with the name of the macro action. Thus, we can avoid calls to initiate decision-theoretic planning, just inserting the new abstract policy. When executing an option in a particular situation we first query our plan library if for the current world situation

an instantiated policy for the option currently to be executed exists. If so, we simply take this policy from the library and execute it. If there does not exist a policy for the option in the current world situation, we have to generate it. We take the situation independent abstract policy for the option and substitute the situation terms with the actual situation. Similarly, we evaluate the value and success probability of the option given the current world situation. With a particular situation we can re-evaluate the precondition axioms of actions, if-conditions, and nondeterministic choices of the abstract policy and obtain one fully instantiated policy which is the same as if we would have calculated it on the fly. To gain computation speed for the next time when the agent wants to execute the option in this particular situation, we store the fully instantiated policy, the value, and the success probability together with the world state. Thus, the next time the option is to be executed in the very same situation, we simply look up the policy without the need to calculate anything at all. We remark that the option concept is different from explanation-based learning (Mitchell et al., 1986), where single examples lead to generalizations. Here we compute the generalization first and then instantiate it when needed, also caching the instance for future re-use.

The READYLOG code to execute a macro action on-line is illustrated below. The predicate `getState` calculates the current world state based on fluent values as described above. The predicate `get_bestPolicy` performs the look-up operation, the predicate `evaluate` assesses the abstract plan tree returning a fully instantiated policy $\pi_s$, which is then executed with `execute`($\pi_s$). The `store` predicate saves the instantiated policy, the value, and the success probability together with the current world situation in the plan library for the next time it is needed. The action $a_{sense}$ is a sensing action which is executed to sense the actual state the agent is in, when trying to execute the option. The logical formula $\varphi_m$ is a condition which checks if the option is executable. This condition can be viewed as a precondition for the option. This precondition is part of the specification of the option and must be provided by the user.

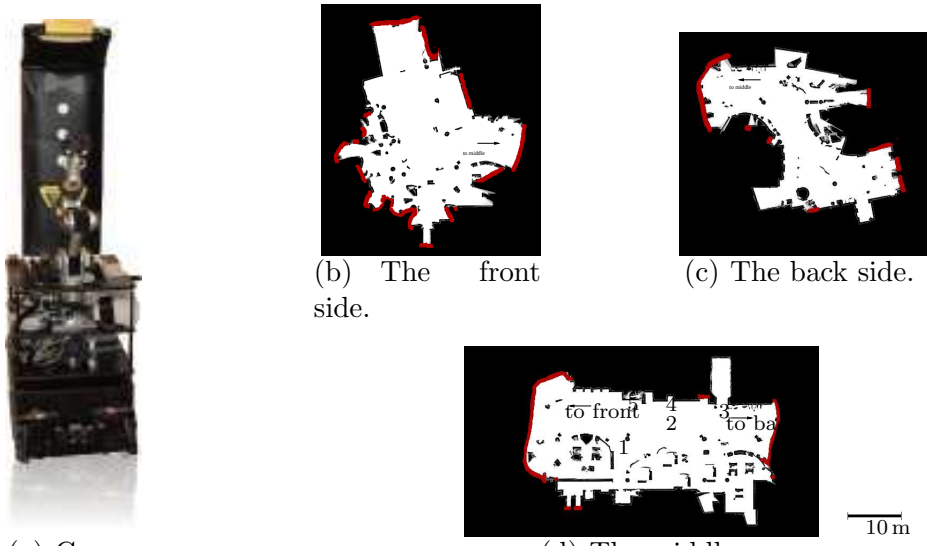
```
getState;
while φ_m do
  if DT Plan Library has entry for current state s then
    get_bestPolicy(s, DT Plan Library, π);
    execute(π_s);
  else
    evaluate(s, AbstractValues, π_s);
    execute(π_s);
    store((s, π_s, v, pr), DT Plan Library);
  endif
  execute(a_sense)
endwhile
```

Next, we give an example of DT macro actions in simulated soccer. For restricting the state space of the soccer domain we make use of a qualitative world model which abstracts from the infinite quantitative state space (Schiffer et al., 2006b). The playing field is divided into grid cells, where each cell of a grid contains infinitely many coordinates. For each of these cells one quantitative representative (the center of this cell) is provided. Several other useful qualitative abstractions for the soccer domain are defined in (Schiffer et al., 2006b). We used two macro action, *outplay opponent* and *create good scoring opportunity* in our test runs in the simulation league. The first macro works as follows: facing attacking opponents, the ball leading agent either dribbles or passes the ball to a teammate. If the macro action chooses the pass, the agent afterwards moves to a free position to be a pass receiver again. The second action aims to *create a good scoring opportunity* to shoot a goal. The agent in ball possession can dribble with the ball if the distance to the opponent's goal is too far. Near the goal the agent can shoot directly to the goal or pass to a teammate that is in a better scoring position. We considered three strategies: (a) using DT planning to cope with the task, (b) using the macro action, but only by evaluating a policy in each step [1], and (c) using the macro action with the plan library that was generated in the last step. Using the planning approach the agent needed 0.1 seconds on average to calculate a policy. With the evaluation strategy (b) only 0.08 seconds are needed. This is a speed-up compared to planning of about 20 %. The time for off-line computations in this example was about 0.02 seconds for each macro. Even taking this pre-processing time into account our macro approach yields reasonable speed-ups. Of course, pre-processing more and more complex macro actions consumes more off-line computation time. But as this time does not need to be spent on-line, this off-line computation time can be justified. The macro action based on the plan library clearly outperforms DT planning. In each test run, for both macro actions, the executing system constantly returns the minimum of measurable time of 0.01 seconds for searching the best plan in the plan library, which corresponds to a mean time saving of over 90%.

## 6 Beyond Robotic Soccer

In the examples above we concentrated on the soccer domain. But we applied Readylog also to other domains. One application domain for Readylog is the service robotics domain (Sect. 6.1), the second very demanding application is the interactive computer game Unreal Tournament 2004 (Sect. 6.2).

[1] Each policy evaluated in this step is stored in the plan library, so we can use this stored knowledge in the next step (c).




(a) Caesar (b) The front side. (c) The back side. (d) The middle

Fig. 8. Our service robot Caesar (RoboCup@Home World Champion 2006, 2007) and a map of a local bank building.

### *6.1 Service Robotics Application*

Our Readylog approach is very suitable for service robotics application because of the easy way, how semantic information about the environment can be integrated into the control of the robot. For example, in our engagement in the RoboCup@Home service robotics competition, we annotate items and furniture of an home environment with semantic tags (Schiffer et al., 2006a). These tags can be easily integrated into the Readylog high-level controller in order to navigate to them or manipulate them. In the following, we show an example of a simple path planner in Readylog.

Figure 8 shows the occupancy grid map of a local bank where the robot operated as a tour-guide. As in the RoboCup@Home scenario, several sight-seeing spots which the robot should guide visitors to can be defined. The nodes of this map are available for Readylog by the fluent *mapNode*, and the relations *childrenOf*(*mapNode*). The program the robot then uses is the procedure *pathPlan* given below. In order to find optimal tours, we used the Readylog's *solve* and *pickBest* statement. Here, the *pickBest* statement chooses the cost-optimal next child node in the map.

**proc** pathPlan($Goal, H$)
  **solve**($H$, *reward_at*(*goal*))
    **while**$\neg$*mapNode* = *goal* **do**
      **pickBest**(*child*, *childrenOf*(*mapNode*), *gotoMapNode*(*child*))
    **endwhile**
**endproc**

The action *gotoMapNode* is in fact a procedure which initiates the robot to drive to the respective coordinate and announce the exhibit. The reward function for the planning task was quite simple. At the goal node the robot receives a high positive reward and zero for all other nodes. When defining a metric on the graph and giving discounts for longer edges one easily can ensure that the robot will take the shortest path to the goal.

**function** $reward\_at(goal)$
$\exists v.mapNode = goal \wedge v = 100 \vee mapNode \neq goal \wedge v = 0)$
**return** $v$

Several other such applications demonstrated the robustness of our approach. It is also very easy to specify new service robotics tasks for applications in RoboCup@Home competitions, for example.

### *6.2 Readylog Game Bots*

Besides service robotics and soccer application we deployed READYLOG also in interactive computer games. In particular, we developed so-called game bots for the interactive computer game UNREAL TOURNAMENT 2004 (Epic Games Inc., 2008), which is a state-of-the-art interactive computer game. The engine itself is mainly written in C++ and cannot be modified. On the other hand, the complete Unreal Script (in the following USCRIPT) code controlling the engine is publicly available and modifiable for each game. For instance, introducing new kinds of game play like playing soccer in teams or the game of Tetris have been implemented on the basis of the Unreal Engine. All this can be defined easily in USCRIPT, a simple, object-oriented, Java-like language which is publicly available. Several different types of game-play or game modes have been implemented for this game. The ones relevant for our work are: (1) *Deathmatch*, (2) *Team Deathmatch*, and (3) *Capture the Flag*. The idea for the first two types of games is to disable as many opponent players as possible without getting disabled oneself. To be successful in this type of game one has to know the world, react quickly, and recognize the necessity to make a strategic withdrawal to recharge. Games where only two players or bots compete against each other in much smaller arenas are especially interesting, as one can compare the fitness of different agents easily in these settings. In the case of *Capture the Flag*, two teams try to score by stealing the flag from the opponent's base. To win such a game the players of a team have to cooperate, to delegate offensive or defensive tasks, and to communicate with each other. This game type is one that rewards strategic defense and coordinated offense maneuvers.


(a) A scene from Unreal Tournament 2004

```
proc agent_dm(Horizon)
  while true do
    solve(Horizon, reward_DM
      if ¬sawOpponent then roam(own)
      else if sawOpponent
      then moveattack(own)) endif ...
      | if itemTypeAvailable(healthPack)
      then collect(healthPack) endif
      | if ¬hasGoodWeapon ∧ itemAvail.(weapon)
      then collect(weapon) endif
    ) /*end solve*/
  endwhile
endproc
```

(b) Readylog code for the game bot

Fig. 9. Readylog bots for Unreal Tournament 2004

In order to develop Readylog game bots, we first had to develop a framework which allows an agent to receive data from the game engine and issue actions like *stop, celebrate, moveto, roam, attack, charge, moveattack*, or *retreat*. The world model our agents have is different from what the built-in bots have available. The built-in bots are omniscient, that is, they have complete knowledge of their environment. Our game bots, on the other hand, can only sense objects visible to them. The world model consists of a large number of fluents from different categories like *bot parameter fluents* (health status, or armor), and *item and bot visibility fluents*, that is, whether or not known items in the environment are visible. More information about the framework can be found in (Jacobs et al., 2005a,b).

Readylog has turned out to be well-suited to this kind of application. To illustrate this we use an excerpt from our actual implementation of the *deathmatch agent* (Fig. 9(b)). Here an agent was programmed which chooses at each action choice point between the outcomes of a finite set of actions. It has the choice between collecting a weapon, retreating to a health item, and so on, based on a given reward function. The main part of the agent is the non-deterministic choice which represents the action the agent performs next. It has the choice between roaming and collecting items, attacking an opponent, or collecting several specific items. The decision which action to take next is performed based on the reward of the resulting state. Note also that the non-deterministic choices are restricted by suitable conditions attached to each choice. This way many choices can be ruled out right away, which helps prune the search space considerably. Our experimental results showed that our Readylog game bots were competitive with the built-in bots, especially in the Capture-the-Flag environment. The reason is that, although the built-in game bots are omniscient, our Readylog agent could react more flexible due to its high-level strategy.

## 7 Conclusion

In this paper, we presented our approach to high-level control of autonomous robots in dynamic domains. We proposed the framework Readylog, a robot programming language suitable to support decision-theoretic planning under uncertainty, macro actions, continuous change, and sensing. The run-time environment combines on-line execution with execution monitoring facilities. We presented examples from the robotic soccer domain, for real robots as well as simulated agents. Besides these soccer applications, we also applied Readylog successfully to interactive computer games as well as tasks in service robotics.

Currently a robot's world model and decision making is only concerned with user-defined missions. In the future we would like to extend this also to the robot's own internal state. This should allow the robot, for example, to recognize and act upon critical states like localization failures or, more generally, to develop a sense of self-awareness, which would help the robot to optimize its behavior.

*Acknowledgments*

We would like to thank the anonymous reviewers for their helpful comments. This work was financially supported by DFG grants LA-747/9-1, 2 and 3.

## References

Beetz, M., 2001. Structured reactive controllers. Journal of Autonomous Agents and Multi-Agent Systems 2 (4), 25–55.

Bonasso, R., Firby, R., Gat, E., Kortenkamp, D., Miller, D., Slack, M., 1997. Experiences with an architecture for intelligent, reactive agents. Journal of Experimental and Theoretical Artifical Intelligence 9 (2-3), 237–256.

Bouguerra, A., Karlsson, L., Saffiotti, A., 2007. Semantic knowledge-based execution monitoring for mobile robots. In: Proc. ICRA-07. pp. 3693–3698.

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., 2000. Decision-theoretic, high-level agent programming in the situation calculus. In: Proc. AAAI-00. AAAI Press, pp. 355–362.

Burgard, W., Cremers, A., Fox, D., Lakemeyer, G., Hähnel, D., Schulz, D., Steiner, W., Thrun, S., 1998. The interactive museum tour-guide robot. In: Proc. AAAI-98. AAAI Press.

Carbone, A., Finzi, A., Orlandini, A., Pirri, F., 2008. Model-based control architecture for attentive robots in rescue scenarios. Auton. Robots 24 (1), 87–120.

De Giacomo, G., Lésperance, Y., Levesque, H., 2000. ConGolog, A concurrent programming language based on situation calculus. Artificial Intelligence 121 (1–2), 109–169.

De Giacomo, G., Levesque, H., 1999. An incremental interpreter for high-level programs with sensing. In: Levesque, H., Pirri, F. (Eds.), Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter. Springer, pp. 86–102.

De Giacomo, G., Levesque, H., Sardiña, S., 2001. Incremental execution of guarded theories. Computational Logic 2 (4), 495–525.

Doherty, P., 2005. Knowledge representation and unmanned aerial vehicles. In: Skowron, A., Agrawal, R., Luck, M., Yamaguchi, T., Morizet-Mahoudeaux, P., Liu, J., Zhong, N. (Eds.), Proc. WI-2005. IEEE Press, pp. 9–16.

Doherty, P., Gustafsson, J., Karlsson, L., Kvarnstrom, J., 1998. TAL: Temporal action logics – language specification and tutorial. Linkoping Electronic Articles in Computer and Information Science 15 (3), 273–306.

Dylla, F., Ferrein, A., Lakemeyer, G., Murray, J., Obst, O., Röfer, T., Schiffer, S., Stolzenburg, F., Visser, U., Wagner, T., 2008. Approaching a formal soccer theory from behaviour specifications in robotic soccer. In: Dabnicki, P., Baca, A. (Eds.), Computer in Sports. WIT Press.

Epic Games Inc., 2008. http://www.unrealtournament.com/.

Ferrein, A., 2008. Robot Controllers for Highly Dynamic Environments with Real-time Constraints. Doctoral dissertation, Knowledge-based Systems Group, RWTH Aachen University, Germany.

Ferrein, A., Fritz, C., Lakemeyer, G., 2003. Extending DTGOLOG with options. In: Gottlob, G., Walsh, T. (Eds.), Proc. IJCAI-03. Morgan Kaufmann.

Ferrein, A., Fritz, C., Lakemeyer, G., 2004. On-line decision-theoretic golog for unpredictable domains. In: Proc. KI-04. Vol. 3238 of Lecture Notes in Computer Science. Springer, pp. 322–336.

Ferrein, A., Fritz, C., Lakemeyer, G., 2005. Using golog for deliberation and team coordination in robotic soccer. KI 19 (1), 24–30.

Finzi, A., Ingrand, F., Muscettola, N., 2004. Model-based executive control through reactive planning for autonomous rovers. Proc. IROS-04, 879–884.

Funge, J., 2000. Cognitive modeling for games and animation. Commununications of the ACM 43 (7), 40–48.

Gelfond, M., Lifschitz, V., 1993. Representing action and change by logic programs. Journal of Logic Programming 17 (2/3&4), 301–321.

Genesereth, M., Love, N., Pell, B., 2005. General game playing: Overview of the AAAI competition. AI Magazine 26 (2), 62–72.

Giacomo, G. D., Iocchi, L., Nardi, D., Rosati, R., 1997. Description logic-based framework for planning with sensing actions. In: Proc. of the 1997 International Workshop on Description Logics,.

Grosskreutz, H., 2000. Probabilistic projection and belief update in the pgolog framework. In: CogRob-00. ECAI-00, pp. pages 34–41.

Grosskreutz, H., Lakemeyer, G., 2001. On-line execution of cc-Golog plans. In: Nebel, B. (Ed.), Proc. IJCAI-01. Morgan Kaufmann.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., Boutilier, C., 1998. Hierarchical Solution of Markov Decision Processes using Macro-actions. In: Proc. UAI-98.

Ingrand, F., Chatila, R., Alami, R., Rober, F., 1996. PRS: A high level supervision and control language for autonomous mobile robots. In: Proc. ICRA-96.

Jacobs, S., Ferrein, A., Lakemeyer, G., 2005a. Controlling unreal tournament 2004 bots with the logic-based action language golog. In: Proc. AIIDE-05.

Jacobs, S., Ferrein, A., Lakemeyer, G., 2005b. Unreal Golog bots. In: IJCAI-05 WS on Reasoning, Representation, and Learning in Computer Games.

Konolige, K., Myers, K., Ruspini, E., Saffiotti, A., 1997. The Saphira architecture: A design for autonomy. JETAI 9 (1), 215–235.

Kvarnström, J., Doherty, P., Haslum, P., 2000. Extending TALplanner with concurrency and resources. In: Horn, W. (Ed.), Proc. ECAI-00. IOS Press, pp. 501–505.

Lakemeyer, G., 1999. On sensing and off-line interpreting in GOLOG. In: Levesque, H., Pirri, F. (Eds.), Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter. Springer, pp. 173–187.

Lemai, S., Ingrand, F., 2004. Interleaving temporal planning and execution in robotics domains. In: Proc. AAAI-04. pp. 617–622.

Levesque, H., Lakemeyer, G., 2007. Cognitive robotics. In: van Harmelen, F., Lifschitz, V., Porter, B. (Eds.), Handbook of Knowledge Representation. Elsevier.

Levesque, H., Pagnucco, M., 2000. Legolog: Inexpensive experiments in cognitive robotics. In: CogRob-00. ECAI-00.

Levesque, H., Reiter, R., Lésperance, Y., Lin, F., Scherl, R., 1997. GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31 (1-3), 59–83.

Lin, F., Reiter, R., 1997. How to progress a database. Artificial Intelligence 92 (1-2), 131–167.

McCarthy, J., 1963. Situations, actions and causal laws. Tech. rep., Stanford University.

McDermott, D., 1991. A reactive plan language. Tech. Rep. YALEU/DCS-RR-864, Yale University, Department of Computer Science.

Mitchell, T. M., Keller, R. M., Kedar-Cabelli, S. T., 1986. Explanation-based generalization: A unifying view. Machine Learning 1, 47–80.

Murphy, R., 2000. Introduction to AI Robotics. The MIT Press.

Muscettola, N., Dorais, G., Fry, C., Levinson, R., Plaunt, C., 2004. Idea: Planning at the core of autonomous reactive agents. In: Proc. 3rd Int. NASA WS on Planning and Scheduling for Space.

Myers, K., 1996. A procedural knowledge approach to task-level control. In: Drabble, B. (Ed.), Proc. AIPS-96. AAAI Press, pp. 158–165.

Pednault, E. P. D., 1989. ADL: exploring the middle ground between STRIPS and the situation calculus. In: Brachman, R., Levesque, H., Reiter, R. (Eds.), Proc. KR-89. Morgan Kaufmann, pp. 324–332.

Pham, H., 2006. Applying DTGolog to large-scale domains. Master's thesis, Department of Electrical and Computer Engineering, Ryerson University, Toronot, Canada.

Pineau, J., Montemerlo, M., Pollack, M. E., Roy, N., Thrun, S., 2003. Towards robotic assistants in nursing homes: Challenges and results. Robotics and Autonomous Systems 42 (3-4), 271–281.

Pirri, F., Mentuccia, I., Storri, S., 2003. The domestic robot - a friendly cognitive system takes care of your home. In: Ambient Intelligence: Impact on Embedded System Design. Kluwer Academic, Boston, pp. 131–159.

Pirri, F., Reiter, R., 1999. Some contributions to the metatheory of the situation calculus. Journal of the ACM 46 (3), 325–361.

Poole, D., 1997. The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94 (1-2), 7–56.

Precup, D., Sutton, R., Singh, S., 1998. Theoretical results on reinforcement learning with temporally abstract options. In: Proc. EMCL-98. Vol. 1398 of Lecture Notes in Computer Science. Springer, pp. 382–393.

Reiter, R., 2001. Knowledge in Action. MIT Press.

Sandewall, E., 1998. Cognitive robotics logic and its metatheory: Features and fluents revisited. Electronic Transactions on Artificial Intelligence 2, 307–329.

Schiffel, S., Thielscher, M., 2007. Automatic construction of a heuristic search function for general game playing. In: Proc. NRAC-07. IJCAI-07.

Schiffer, S., Ferrein, A., Lakemeyer, G., 2006a. Football is coming home. In: Chen, X., Liu, W., Williams, M.-A. (Eds.), Proc. PCAR-06. University of Western Australia Press.

Schiffer, S., Ferrein, A., Lakemeyer, G., 2006b. Qualitative world models for soccer robots. In: Wölfl, S., Mossakowski, T. (Eds.), Qualitative Constraint Calculi, Workshop at KI 2006, Bremen. pp. 3–14.

Soutchanski, M., 2001. An on-line decision-theoretic Golog interpreter. In: Nebel, B. (Ed.), Proc. IJCAI-01. Morgan Kaufmann.

Strack, A., Ferrein, A., Lakemeyer, G., 2006. Laser-based localization with sparse landmarks. In: Proc. RoboCup Symposium 2004. Springer, pp. 569–576.

Thielscher, M., 1998. Introduction to the Fluent Calculus. Electronic Transactions on Artificial Intelligence 2 (3–4), 179–192.

Thielscher, M., 2000. Representing the knowledge of a robot. In: Cohn, A., Giunchiglia, F., Selman, B. (Eds.), Proc. KR-00. Morgan Kaufmann, pp. 109–120.

Thielscher, M., 2005. FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming 5 (4-5), 533–565.

Tira-Thompson, E., 2004. Tekkotsu: A rapid development framework for robotics. Master's thesis, Robotics Institute, Carnegie Mellon University.