Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines

Am Fachbereich Informatik der
Technischen Universität Darmstadt
eingereichte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Max Risler
(geboren in Karlsruhe)

Referenten der Arbeit: Prof. Dr. Oskar von Stryk

Prof. Dr. sc. nat. Hans-Dieter Burkhard
(Humboldt-Universität zu Berlin)

Tag der Einreichung: 26.3.2009
Tag der mündlichen Prüfung: 15.5.2009

D17

Acknowledgements

This thesis was created during my time as a research assistant at *Simulation, Systems Optimization, and Robotics Group* at the *Department of Computer Science* of *Technische Universität Darmstadt*. Parts of my research have been supported by the *German Research Foundation (DFG)* within the special priority program 1125 on *Cooperating Teams of Mobile Robots in Dynamic Environments*.

I would like to thank my supervisor Prof. Dr. Oskar von Stryk for the opportunity to work on this interesting and challenging field. I would also like to thank him for his excellent scientific mentorship and for many valuable suggestions and fruitful discussions. He always had time for me when his advice was required.

I want to thank Prof. Dr. Hans-Dieter Burkhard for the successful long-time collaboration in the joint *RoboCup* team *GermanTeam* and for accepting the Korreferat for my thesis.

I would also like to thank all of my colleagues, students, and team members of our *RoboCup* teams. All of you contributed a lot to the success of this work and you made my time in Darmstadt always very enjoyable.

Thanks to Martha Erhard and Marc Eckart for helping me putting the final touches to this thesis.

Finally I want to say thank-you to those that always supported me in so many different ways: my parents, my sister Christiane, and Barbara Rothe.

Darmstadt, 2009 Max Risler

Contents

Zusammenfassung

Komplexe Verhaltenssteuerungen für kooperative Multiagentenanwendungen in dynamischen Umgebungen, wie sie in vielen realen Anwendungen auftreten, stellen eine große Herausforderung dar. Es werden pragmatische und effiziente Methoden benötigt um Agentenverhalten zu implementieren, die in der Lage sind, mit den erforderlichen Echtzeitanforderungen, der unvollständigen bzw. verrauschten Wahrnehmung der Umgebung und der Unvorhersehbarkeit dynamischer Umgebungen umzugehen.

Diese Arbeit beschäftigt sich mit der Verhaltenssteuerung autonomer Agenten, d.h. mit dem Teil einer Agentensoftware der jegliche nicht triviale Entscheidungsfindung umfasst, die benötigt wird um komplexes autonomes Verhalten zu realisieren.

Ziel dieser Arbeit ist, eine Verhaltenssteuerungsarchitektur für autonome Agenten zu entwickeln, die geeignet ist, um komplexe reale Roboteranwendungen in einer komfortablen und zeiteffizienten Art und Weise zu erstellen. Die gesuchte Architektur soll einerseits dem Entwickler die Möglichkeit bieten das genaue Verhalten eines Agenten in bestimmten Situationen explizit angeben zu können und andererseits Skalierbarkeit aufweisen, um in der Lage sein mit der sehr großen Komplexität von Verhaltenssteuerungen umzugehen, die in der Regel bei allen nicht trivialen realistischen Anwendungen auftreten. Es darf keinerlei Einschränkungen an die Arten der Aktionsauswahlmechanismen geben, die in der Verhaltenssteuerung zum Einsatz kommen. Beispielsweise muss es möglich sein, sowohl reaktives als auch deliberative Verhalten zu kombinieren. Eine weitere Anforderung ist die Möglichkeit zur nebenläufigen Ausführung einzelner Teilverhalten. Es soll nicht nur die Auswahl diskreter Aktionen, sondern auch die Erzeugung von kontinuierlichen Ausgabesignalen, unterstützt werden. Ausserdem darf es keine Einschränkungen an das Anwendungsgebiet sowie an die verwendete Hardwareplattform geben.

In dieser Arbeit werden methodische Weiterentwicklungen vorgestellt, die an einer auf hierarchischen Zustandsautomaten basierenden Verhaltenssteuerungsarchitektur vorgenommen wurden. Es wird die hieraus resultierende erweiterte Version der Verhaltensbeschreibungsarchitektur und -sprache *Extensible Agent Behavior Specification Language*(XABSL) vorgestellt. Neben Verbesserungen in der Benutzbarkeit umfassen die Erweiterungen unter anderem Möglichkeiten zur nebenläufigen Verhaltensausführung sowie verbesserte Möglichkeiten der Umsetzung kontinuierlicher Verhaltensausgaben. Anhand verschiedener Anwendungsbeispiele wird dargelegt, dass die resultierende Architektur die beschriebenen Entwurfsanforderungen erfüllt.

XABSL ermöglicht die bequeme Entwicklung von Verhaltenssteuerungen autonomer Agenten auch für sehr große und komplexe reale Roboteranwendungen. Zustandsbasierte Techniken erlauben den Umgang mit Unsicherheiten in hoch dynamischen Umgebungen. Die Komposition von Verhaltensmodulen, die auf Zustandsautomaten basieren, zu komplexen Hierarchien gibt die Möglichkeit zur Wiederverwendbarkeit einzelner Teilverhalten in unterschiedlichen Kontexten und erleichtert somit die Entwicklung skalierbarer, komplexer Verhalten.

Die primäre Testumgebung, in der die Verhaltenssteuerungsarchitektur angewandt wird, ist das Anwendungsszenario Roboterfußball. XABSL wurde unter Mitwirkung des Autors, der seit 2004 die Weiterentwicklung übernommen hat, ab 2002 in der Roboterfußballmannschadt *GermanTeam*, welches in der *RoboCup Four-Legged League* antritt, entwickelt und eingesetzt. XABSL findet stetig zunehmende Verbreitung. Es wird von vielen Teams in verschiedenen Ligen des *RoboCup* eingesetzt. Das *GermanTeam* konnte 2004, 2005 und 2008 Weltmeister in der *RoboCup Four-Legged League* werden.

Auch wenn es dort die größte Verbreitung findet ist XABSL nicht auf die Anwendung Roboterfußball beschränkt. Es gibt keine Architektur- oder Sprachelemente, die für die Fußballanwendung spezifisch wären. Die Architektur, die Sprache und die Laufzeitumgebung (*XabslEngine*) sind anwendungs- und plattformunabhängig und können somit auf beliebigen Agentensystem zum Einsatz kommen. Ein Beispiel einer erfolgreichen Anwendung außerhalb der Roboterfußballdomäne findet sich in der Verhaltensprogrammierung des *Darmstadt Rescue Robot Teams*.

1 Introduction

Complex behaviors for cooperative multi-agent applications pose a challenging task in highly dynamic environments as they are encountered in many real-world applications. Efficient methods are required for programming agent behaviors that are able to cope with necessary real-time requirements, only partial or noisy observability of the environment, and the unpredictability of dynamic environments.

1.1 Agent Behavior Control

This work focuses on the part of agent software which contains any nontrivial decision-making functions required for realizing complex autonomous behaviors. It is assumed that the agent receives some form of input, either directly from external sources, such as data from sensory perceptions, or preprocessed information items generated from previous inputs, such as estimations of the current state of the environment. These inputs are then used for deciding which actions are selected by the agent. This form of action selection is denoted *behavior control*. Neither the actual input generation which could include the world or belief modeling of an agent nor how execution of actions is accomplished, e.g. the actuator control of a robotic agent, are in the scope of this work.

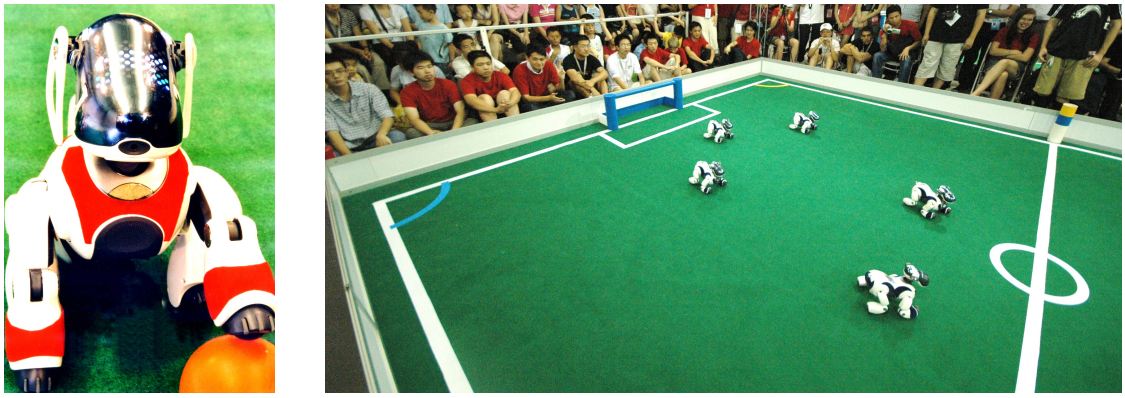
Furthermore, it is assumed that behavior control is to be executed at discrete time steps. Behavior control can either be triggered at fixed time intervals or it is triggered by certain events, e.g. as soon as new input data is available from sensors or from sensor processing. Behavior control execution could also be coupled to the cycle rate of one of the input sources, e.g. if the main sensor of a robot is a camera system, behavior control is executed if and only if a new image was processed.

If action and perception cycles are asynchronous and action selection is triggered during the perception cycle it is necessary to store the results from action selection in order to be evaluated in the action cycle. Thus, the traditional AI paradigm of using perceptual input to trigger actions is applicable in the asynchronous case as well.

When multi-agent applications are investigated, communication with cooperating agents can be treated as additional inputs and outputs to behavior control. It is assumed that adequate communication channels are provided and incoming messages from other agents are provided to behavior control which generates outgoing messages.

1.2 Application Scenario: RoboCup Soccer

A test bed for cooperative multi-agent applications can be found in the *RoboCup* [53] robot soccer domain. The *RoboCup* initiative was founded in 1997. Its aim is to support research in artificial intelligence and robotics by providing a standard problem which serves as a benchmark for a wide range of developments and technologies.



(a) (b)

Figure 1.1: A *Sony Aibo ERS7* robot and the soccer field of the *Standard Platform League* from 2008.

In *RoboCup* there are different leagues that deal with the problem of autonomous robot soccer on different levels.

In the *Simulation Leagues* soccer games are run with 11 simulated players per team on a simulation server. The human team members program agents running in the simulated environment. Unlike the other leagues teams do not have to cope with real robotic hardware. Still, the simulation includes certain limitations in the capabilities of the soccer players in order to simulate reasonably realistic games.

In the *Small Size Robot League* teams consist of five small (at most 18 cm diameter and 15 cm height) fast-moving wheeled robots. Teams have the ability to observe the game with a global vision system using overhead cameras and control the robots from an external computer. Therefore, the robots only need to have onboard computational power for motor control. Behavior control takes places globally controlling the complete team at once.

In contrast, the other robot leagues use robotic agents that are completely autonomous, using local sensing and vision systems and no external computational power.

In the *Middle Size Robot League* teams consist of five autonomous wheeled robots (no larger than 50 cm $\times$ 50 cm $\times$ 80 cm). There are no restrictions on onboard sensing systems. Most teams use omnidirectional vision systems.

In the *Humanoid League* the autonomous robots have to apply biped locomotion. Also there are certain restrictions on the design of the robots requiring human-like proportions and sensing capabilities.

In the *Standard Platform League* all teams have to apply the same robotic platform. This alleviates the teams from having to design their own robot hardware. Instead teams have to focus on the software of their autonomous robots. From 1998 until 2008 the *Sony Aibo ERS110*, *ERS210*, and *ERS7* four-legged robots [34] as shown in Figure 1.1(a) were used as the standard platform. In 2008 the *Aldebaran Nao* humanoid robot [1] was introduced as a new standard platform after the production of *Aibo* was discontinued in 2006. As until 2007 the *Aibo* was the only platform available, the league was formerly called *Four-Legged Robot League*.

The author of this work is a founding member and the team leader of the *GermanTeam* which since 2001 has been active in the *Four-Legged Robot League* (now called *Standard Platform League*). The *GermanTeam* is one of the most successful teams in its league, becoming world champion three times in 2004, 2005 and 2008. The *GermanTeam* is a joint project consisting of researchers and students from the German universities *Technische Universität Darmstadt*, *Humboldt-Universität zu Berlin*, *Universität Bremen*, and until 2005 *Universität Dortmund*.

The *Sony Aibo ERS7* robot has three controllable degrees of freedom in each of its four legs and in the head. The head contains the main sensor of the robot: a CCD camera recording 30 images per second at a resolution of 208 × 160 pixels and a horizontal opening angle of 55 degrees. Further sensors of the robot are two distance sensors in the head and the chest, a three-axes-accelerometer, two microphones in the head, and various touch sensors. The computing unit contains a 576 MHz MIPS processor and 64 MB of RAM. In the last *Aibo* competition held in the *Standard Platform League* in 2008, games were played by teams of five robots each on a playing field of 6.9 m × 4.6 m size (cf. Figure 1.1(b)). All of the objects that need to be detected by the robots' cameras are color coded: the ball is orange, the field is green, field lines are white, the goals and additional localization landmarks are sky-blue and yellow, and the robots are marked with colored jerseys, one team in red and the other team in blue. The robots have to act completely autonomously. There is no external computing or remote control. The only exceptions are referee commands which are transmitted via wireless network, e.g. in order to signal goals and penalties.

In contrast to the leagues based on wheeled robots, both the *Humanoid League* and the *Standard Platform League* are characterized by a high motion complexity. Due to the complex legged locomotion the choice of possible actions is limited and strongly depends on the current situation. For instance, unlike a driving robot a walking robot might not be able to change its walking direction at once, because it would lose balance and fall over when trying to do so. Also there is a large amount of uncertainty in the execution of actions, due to foot slippage, which is not necessarily present in wheeled locomotion. Besides, behavior control has to cope with a incomplete and noisy world model as perceptions can be sparse and inaccurate. Another aspect influencing the behavior control complexity in both leagues is directed vision with relatively small camera opening angles. In order to gain sufficient information from the camera, coordination of head and leg movements with vision requirements is necessary.

During the active years of the *GermanTeam* there was a steady development leading to increased quality and reliability of available world model information and also resulting in improved motion capabilities. Some recent improvements are described in [12, 39, 41, 48, 49, 84]. Details on the algorithms that have actually been applied in robot soccer competitions can be found in [11, 87, 88, 92]. For a complete list of publications of the *GermanTeam* see [86].

In order to be able to quickly tap the full potential resulting from given perception and motion realizations, it is of paramount importance to be able to adjust behavior control action selection mechanisms with minimal effort in order to reflect changed premises. Therefore, a behavior control architecture is required which enables rapid development of application implementations.

Figure 1.2: The top three robotic vehicles from *Carnegie Mellon University*, *Stanford University*, and *Virginia Tech*, which have finished the *DARPA Urban Challenge* (taken from [47]).

1.3 Further Application Examples

The *RoboCup* initiative does not only consist of the soccer scenario. Another application which serves as a benchmark for robot teams is disaster rescue. The *RoboCupRescue* project involves potentially very large and heterogeneous teams of agents operating in a hostile environment. There are competitions with simulated as well as real robots that are intended to promote research in this domain. For the real robot competitions the task is to explore an arena, which is modeled after a disaster site, which contains stairs, platforms, and rubble. The robots are supposed to detect and localize possible victims represented through simulated life signs, such as heat, waving hands, or shouting noises. Robots can operate either by remote control or autonomously. The teams have to produce a map showing the locations of detected victims. The focus of this competition is more on research issues such as the design of powerful and highly maneuverable robotic hardware, simultaneous localization and mapping, and human-robot interaction, and less on autonomous operation. Therefore, the behavior control required for these robots usually is less complex than behavior control for robot soccer. Still, *RoboCupRescue* also provides an interesting test bed for behavior control for autonomous agents.

Another prominent example of real-world robot applications is autonomous driving. In 2007 the *Defense Advanced Research Projects Agency (DARPA)*, the research organization of the United States Department of Defense, organized the *Urban Challenge*: a competition for the development of an autonomous ground vehicle operating in an urban environment. The task included driving autonomously on a 96 km course while obeying traffic regulations and handling traffic. While this application definitely is more difficult than the other quoted applications, similar to the previous example, the main difficulty does not necessarily lie in the complexity of the required behavior control. The main difficulties rather arise from the huge amount of uncertainty caused by having to operate in a natural, unstructured environment. The behavior control implementations even the most successful teams applied in this competition were relatively small for instance compared to behavior control implementations in successful robot soccer applications. The team *VictorTango* from *Virginia Tech* which placed third in the competition with the vehicle

shown on the right in Figure 1.2, developed a hierarchical approach for programming high-level driving behaviors, which is implemented in *LabVIEW* [22, 47]. Their hierarchical behavior decomposition consists of only about ten independent behavior modules while the soccer behavior implementations under investigation in this work often are composed out of about hundred or more behavior modules (cf. Table 6.2 in Section 6.1). Team *AnnieWAY*, one of two German teams that qualified for the final round of the challenge, also applied hierarchical state machines for high-level behavior control, which they implemented in C++ using a free library for statecharts [38, 50].

1.4 Contents and Contributions

The aim of this work is to develop a behavior control architecture for autonomous agents that is suited for realizing complex real-world robot applications in a comfortable and time-efficient manner. The desired architecture should on the one hand provide the developer with the possibility to specify explicitly what exactly the behavior should look like in certain situations, while on the other hand the architecture must be capable of scaling up to the huge complexity required for most non-trivial realistic applications. There should be no restrictions on the kind of action selection mechanism to be realized. For instance, reactive behaviors should be realizable as well as deliberative behaviors, also with the potential to be executed concurrently. Not only discrete action selection should be supported, but the generation of continuous outputs should also be possible. Also there must be no restrictions on the application domain and the hardware platform.

The structure of this thesis is as follows: Chapter 2 gives an overview of the state of research on agent behavior control as it was defined in this chapter. Some selected methodologies for programming agent behaviors are presented. It is focused on examining whether the described solutions are suited for the kind of pragmatic programming of complex real-world robot applications that are in the focus of this work. None of the described architectures fully meets all of the desired properties mentioned above.

In Chapter 3 the most important properties required by a behavior control architecture is required to have in order to be suited for programming complex real-world applications are specified briefly.

Another behavior control architecture is the *Extensible Behavior Specification Language* (XABSL), which was developed by Martin Lötzsch with the collaboration of the author and other members of the *GermanTeam* [65, 67]. It has been applied for realizing robot soccer applications since 2002. Since 2004, the author is the sole maintainer and main contributor to the XABSL architecture and its development and implementation. In Chapter 4 the XABSL architecture and its extensions are described which were added to the first XABSL behavior control architecture in order to meet the design goals stated in Chapter 3.

Chapter 5 describes the resulting behavior control architecture developed in this thesis which is based on hierarchical finite state machines. The description also considers design motivations that led to certain decisions during the development of the architecture.

A summary of the resulting behavior implementations facilitated by this work is given in Chapter 6. Different applications that have been realized using XABSL are presented.

It is demonstrated how the developed architecture fulfills the design goals by discussion of various of application examples.

Chapter 7 gives concluding remarks.

2 State of Research

The focus of this work is on finding an efficient solution for engineering decision making of software agents for real-world applications. Approaches from classical symbolic and knowledge based AI [95] have been researched intensively for many years. But it is a difficult task to cope with the complexity of the system by means of logic when agents have to deal with noisy sensor readings, unpredictable dynamics of the world, and uncertainty of actions. As Gat [37] remarked: "Elevator doors and oncoming trucks wait for no theorem prover."

Expressing scepticism towards traditional AI research in "block world" domains, researchers came up with the *behavior based* paradigm [7, 19]. In these biologically inspired approaches direct sensor-actuator couplings control the overall behavior of an agent. To obtain more complex behaviors, several of such behavior units or modules are combined continuously [6], competitively [71], in layers [17], or state based. Although impressive behaviors have been realized with such approaches, it still needs to be shown how to scale up these systems.

Many researchers in the field of autonomous agents try to minimize the role of the designer. Some of them propose general action selection mechanisms that "automatically" choose between different alternatives. For example, alternative behaviors could provide an activation level based on their utility in the current state of the environment. An automated selection mechanism could choose the behavior with the highest activation. Other researchers build systems that aim for learning complex hierarchical interactions with the environment.

These approaches are definitely moving in the right direction towards true machine intelligence, but there are several problems when applying the current state of the art in more complex applications such as robotic soccer. First of all, scalability and extensibility are key issues: adding new behaviors to existing ones is often difficult as behaviors influence each other and the utility estimations of all other behaviors have to be adapted in order to integrate a new behavior. Additionally, it is often not enough that the agents exhibit meaningful and versatile behaviors – developers sometimes just want to specify explicitly what the agents shall do in certain situations. This can be done by a time-consuming tuning of utility measures or by adapting the learning problem. The problem with that is that explicit instructions on what to do in particular situations are hidden implicitly in the specification of the environment, in the action selection algorithm, or in the reward function of a learning algorithm. Due to such difficulties developers often do not use any of these approaches when they program autonomous agents to perform specific tasks. Instead they hand-code the behaviors in native programming languages like *C++*, in scripting languages such as *Perl* or in graphical development environments such as *LabVIEW* (e.g. in [22]) and *MATLAB Simulink*.

In the remainder of this section some of the existing solutions for agent behavior programming are presented.

2.1 Golog

Golog is an example of a classical symbolic and knowledge based AI approach for agent description, where "planning", i.e. generating appropriate actions in order to carry out a given task, is reduced to problem solving [64]. It is a logic programming language based on the *Situation Calculus* [73]. Analogous to *Prolog* programs in *Golog* are interpreted by a theorem prover. This requires symbolic representations of the world and its static and dynamic constraints as well as of the impact of actions on the environment.

As described above a pure logic approach is not very well suited for realizing real world applications. In the case of *Golog* an extension has been proposed which adds concepts such as execution failures and timeouts and features for handling sensor data and user inputs, thus making *Golog* practical for some applications [43].

2.2 PDDL

Similar to *Golog*, *PDDL (The Planning Domain Definition Language)* [75] is a programming language for defining classical planning problems. Similarly, it is not intended for programming agent behavior directly, but rather for specifying planning problems by defining goals, actions, effects, and axioms as the input for a planner.

2.3 Behavior Language

One of the first languages designated for behavior specification is the *Behavior Language* by Brooks [18] which is based on an extended version of the *subsumption architecture* [17]. The subsumption architecture is a hierarchical architecture which uses layers with different levels of competence. A complex system is built bottom-up from lower layers providing basic functionality to higher-level layers adding complexity on top of the lower layers. Layers are running unaware of the other layers above which can interfere with their data paths. The Behavior Language is a rule-based parallel programming language. Each behavior is specified through a set of rules. Communication between different behaviors is realized by message passing. Rule sets are written in a subset of Lisp and can be compiled into a network of finite state machines augmented with timers. These state machines can be compiled directly into Assembler code for certain processors or into Common Lisp programs.

The Behavior Language is one of the first approaches for behavior programming which applies a hierarchy of finite state machines.

2.4 Reactive Plan Language (RPL)

Another language for describing high level reactive plans is the *Reactive Plan Language (RPL)* [74][13]. The programming language *RPL* is very similar to *LISP* with added control structures such as sequencing, concurrent execution, loops, and subroutine calls. Concepts such as interrupts and monitors are provided in order to synchronize parallel actions. *RPL* has been successfully applied on the robotic museum tour-guide *Minerva* [100].

As programs are based on the functional and recursive notation of *LISP*, *RPL* programs are suited for letting a planner reason about how well a program will perform at a certain task.

Therefore, this approach, similar to those described before, aims at generating actions through automatic problem solving. While this is definitely a very interesting field, this thesis rather aims at the efficient programming of complex agent behaviors, as it can be found in the architectures described in the following sections.

2.5 Configuration Description Language (CDL)

The *Configuration Description Language* (CDL) [70] is part of the *MissionLab* system and is inspired by the theory of societies of agents [76]. Complex behaviors are assembled out of subordinated primitive agents in three different methods of coordination:

- Competitive: According to some metric the active agent is selected out of a subset of possible agents.
- Temporal Sequencing: The active agent is determined by a finite state machine. Each state of the state machine specifies an subordinated agent which is in control as long as the respective state is active.
- Cooperative: The output of the agent is a combination of the outputs of the subordinated agents, for instance a weighted vector summation. This allows the combination of the results of different behaviors in a continuous manner.

Assemblage agents can be used as primitive agents in other assemblages thus allowing the reuse of behaviors in different contexts and the construction of complex hierarchical agent configurations.

There is a graphical editor for creating and modifying agent assemblages.

Another interesting feature of this architecture is the support of reinforcement learning behaviors. In a specific type of learning assemblages Q-Learning (cf. Section 2.11.1.2) is applied in order to coordinate a set of behavior assemblages [72]. The developer selects a number of relevant state variables and a set of behaviors and defines state and action combinations in which the agent is rewarded. A policy for selecting one of the actions depending on the input state is generated using the Q-Learning algorithm.

2.6 COLBERT

COLBERT is a language for reactive behavior control based on finite state machines [58]. State machines are implicitly defined via procedure specifications. The architecture facilitates hierarchical and concurrent execution of state machines. It was developed for reactive control in the Saphira architecture [59]. *COLBERT* is based on a subset of ANSI C with extensions for robot control. An interpreter can execute source code directly, thus enabling runtime monitoring and debugging. For performance reasons it is also possible to compile the source code to native C code. Statements in *COLBERT* are mapped to states of finite state machines. The language provides additional features such as support

```
act approach()
{
  int x;
  start patrol(-1) timeout 300 noblock;
 checking:
  if (timedout(patrol) || sfStalledMotor(sfLEFT))
    fail;
  x = ObjInFront();
  if (x > 2000) goto checking;
  suspend patrol;
  move(x - 200);
  succeed;
}
```

Figure 2.1: An example of a COLBERT procedure (taken from [58]).

for implementing timeouts or sending signals in order to control the execution of state machines.

Figures 2.1 and 2.2 show an example behavior in COLBERT and how the same behavior could be addressed using hierarchical state machines as described within this work.

2.7 Petri Net Plans

A formal approach using Petri nets for modeling robot behavior can be found in [107]. This approach and the architecture presented in this work have several common features such as hierarchical decomposition of complex behaviors, concurrent execution of partial behaviors, and support for multi-robot cooperation. Modeling behavior with hierarchical finite state machines has a similar expressiveness while in the author's opinion it is more intuitive since it utilizes more compact behavior descriptions. A property of the Petri nets formalism is the possibility of the analysis and verification of certain properties of the specified behaviors. Similar formal analysis can also be done using hierarchical state machines [3].

Figures 2.3(a) and 2.3(b) give an exemplified comparison of Petri nets and state machine specifications in XABSL (cf. Section 4) for a small behavior routine. In this example the behavior of a robot soccer player is modeled which is supposed to search and approach a ball. To solve this task, the partial or primitive behaviors *seekBall*, *approachBall*, and *trackBall* are applied, which respectively let the robot search for the ball, move towards the ball, and track the ball using a camera located at the head of the robot. In both versions the robot will first assume that the position of the ball is unknown and search for it. When it finds it, the robot will concurrently move towards the ball and track the ball with the camera until it arrives at the ball, and thus the target state of this behavior is reached.

```
option approach
{
    initial state patrol
    {
        decision
        {
            if (state_time > 300) goto fail;
            else if (stalled_motor(motor = left)) goto fail;
            else if (obj_in_front > 2000) stay;
            else goto move;
        }
        action
        {
            patrol(n = -1);
        }
    }
    state move
    {
        decision
        {
            if (action_done) goto succeed;
            else stay;
        }
        action
        {
            move(x = obj_in_front - 200);
        }
    }
    target state succeed
    {
        action {}
    }
    target state fail
    {
        action {}
    }
}
```

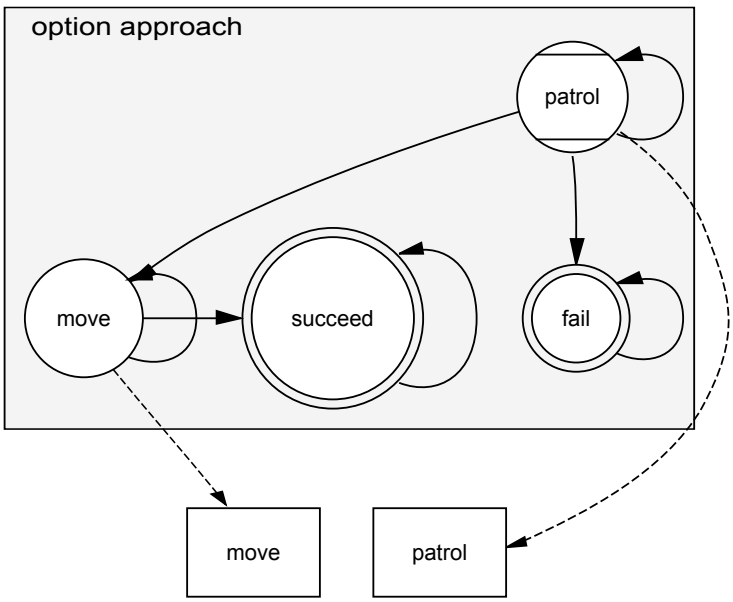
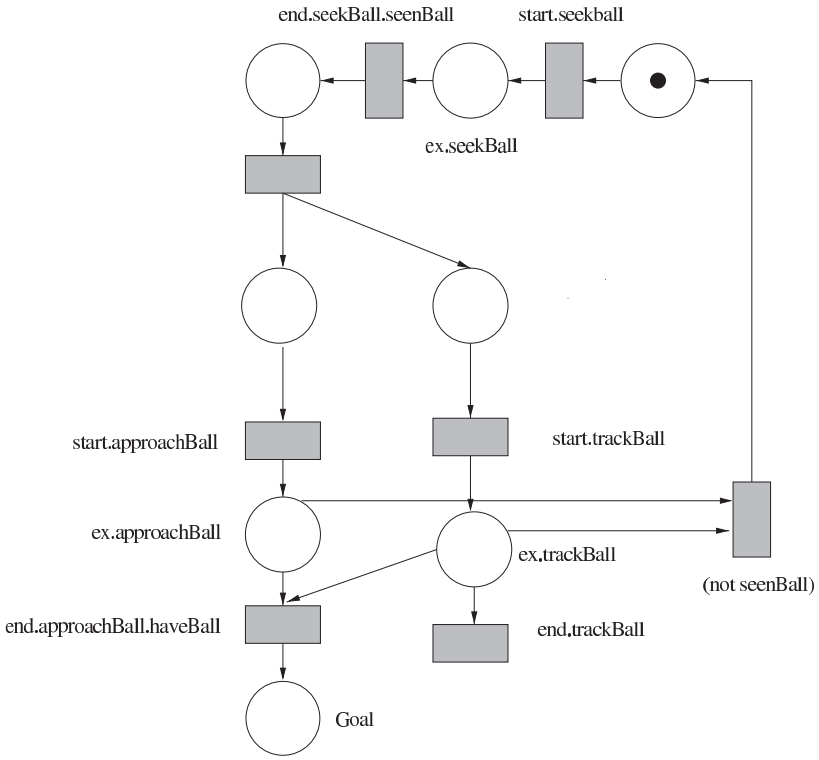
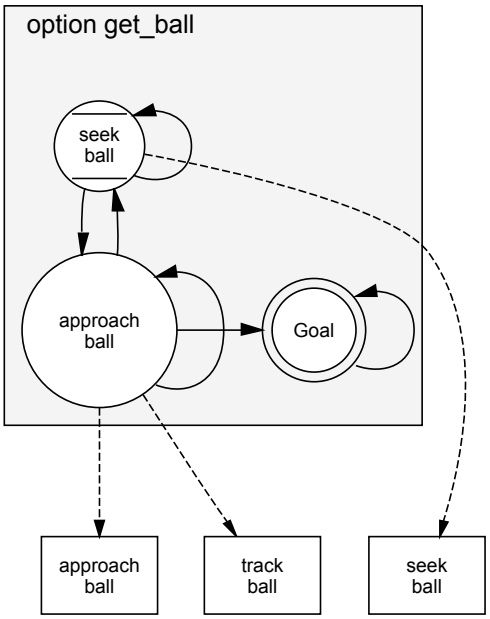


Figure 2.2: A possible translation of the COLBERT example to XABSL (cf. Section 4) and its visualization.



(a) An example behavior modeled with Petri nets (taken from [107]).



(b) A possible adaptation of the Petri nets example using concurrent hierarchical state machines in XABSL (cf. Section 4).

Figure 2.3: Comparison of Petri nets and hierarchical state machines for an example behavior for searching and approaching a ball.

2.8 Statecharts

Harel statecharts constitute a common formal approach for the specification of hierarchical state machines. They were introduced by Harel in 1987[44]. Statecharts enjoy widespread usage as they are part of the *Unified Modeling Language(UML)*[79]. Statecharts do not present an actual architecture that can be applied to implement agent behavior. Instead it is rather a method for the formal description of the behavior of software (or robotic) agents.

2.9 Hybrid Automaton Language (HAL)

Another comparable formal approach for multi-agent behavior specification which is based on a combination of hierarchies of hybrid automata [2] and UML statecharts [44] called Hybrid Automaton Language (HAL) is described in [35, 78]. It is based on an earlier version which was also applying UML statecharts but did not yet include continuous behavior aspects [5, 77]. Similar to the previous approach it focuses primarily on formal analysis and verification and supports model checking. The advantage of using hybrid automata for modeling behaviors is that continuous variables and their dependencies can be integrated directly.

A translator for creating XABSL specifications (cf. Section 4) from hybrid automata specifications has been developed [93].

Figure 2.4 shows an example from a robot soccer application.

2.10 Double Pass Architecture (DPA)

An interesting behavior control architecture can be found in the *Double Pass Architecture (DPA)* [14, 20, 21]. It is based on hierarchical finite state machines and on *BDI* concepts [16, 105] (*BDI* = belief, desire, intentions). It distributes the actual decision making into two execution passes: deliberation and execution. The deliberation pass is responsible for selecting intentions which can be seen as long-term goals of the agent. The executor pass performs potentially time critical decision making in accordance with the selected intentions of the agent.

In applications where efficient execution times are required – which is the case in most real robot applications – and where decision making includes computationally expensive operations, *DPA* can be a very useful alternative for behavior control.

2.11 Machine Learning Approaches

Machine learning approaches can provide effective means in order to control agent behavior to solve a given task. Using techniques such as hierarchical reinforcement learning these approaches have also been shown to be scalable for large and complex problems [8].

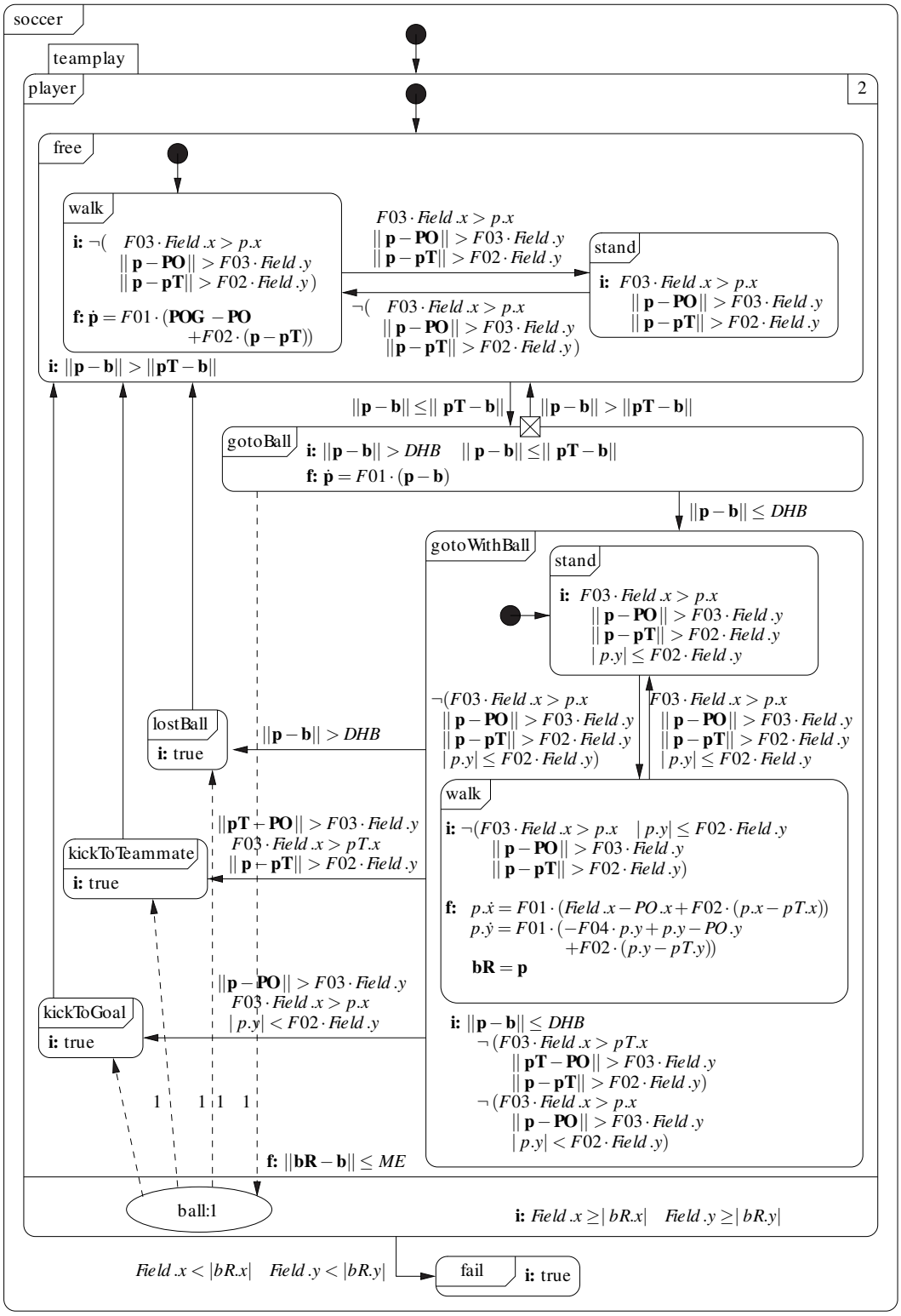


Figure 2.4: A robot soccer example modeled as hybrid state machine (taken from [35]). In this example two robot soccer players cooperate. One robot will go to the ball and kick it to the goal or pass it to the other player, while the other player positions itself strategically.

Nevertheless, especially when dealing with dynamic environments, sometimes the need arises for a developer to specify explicitly what actions an agent should select in certain situations. When using machine learning, such explicit directives can often only be incorporated by adapting reward functions or by modifying the learning problem. Furthermore, real-world problems usually have high dimensional continuous state spaces which often exceed the possibilities of machine learning methods. Because of these difficulties, in many real-world autonomous robot applications such approaches prove to be inappropriate and instead, agent behaviors are programmed manually in standard programming languages.

This work also investigates how to apply machine learning approaches in a beneficial way in complex dynamic real-world robot scenarios by combining them with explicit behavior programming, while utilizing hierarchical behavior decomposition.

2.11.1 Reinforcement Learning

In Reinforcement Learning an agent interacts with its environment continually by selecting actions which result in responses from the environment [98]. The environment also provides reward values to the agent. The agent's goal is to maximize the reward over time.

At each discrete time step t, the agent is provided with a representation of the current state of the environment $s_t \in S$, where S is the set of possible states. The agent selects an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of possible actions in state s_t. In the next time step, as a consequence of the action a_t, the agent will receive reward r_{t+1} and perceive the state s_{t+1}.

In order to select its action in each time step, the agent applies a mapping from state to action selection probabilities. This mapping is the policy π_t. Under a stochastic policy the probability of selecting action a in state s is given by $\pi_t(a, s)$. A reinforcement learning method specifies how π_t is modified in order to maximize the received reward.

Usually finding the optimal policy π^* involves estimation of the action-value function Q^π for policy π. $Q^\pi(s, a)$ denotes the excepted sum of rewards received when taking action a in state s and thereafter following policy π:

$$Q^\pi(s, a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\},$$

where $E_\pi\{\}$ means that the argument is evaluated under the assumption that the policy π is being followed. The parameter γ, $0 \leq \gamma \leq 1$, is the discount rate, representing the present value of future rewards.

Given Q^π it is easy to improve the policy π towards π^* simply by making it greedy in respect to Q^π which means to always select the action which maximizes $Q^\pi(s, a)$. Often instead of always selecting the best action an ϵ-greedy policy is applied which differs from the greedy policy by having a small probability ϵ of selecting a random action instead of the greedy action. This is done to provide ongoing exploration which usually is required to assure convergence of π to the optimal policy.

One specific class of Reinforcement Learning is Temporal-Difference Learning with the basic idea of using immediate rewards received during one time step in order to improve

```
Initialize Q(s,a) arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy from Q (e.g. ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy from Q (e.g. ε-greedy)
    Q(s,a)←Q(s,a) + α[r + γQ(s',a') − Q(s,a)]
    s←s'; a←a'
  until s is terminal
```

Figure 2.5: The Sarsa algorithm (taken from [98]).

```
Initialize Q(s,a) arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy from Q (e.g. ε-greedy)
    Take action a, observe r, s'
    Q(s,a)←Q(s,a) + α[r + γmax_a' Q(s',a') − Q(s,a)]
    s←s'
  until s is terminal
```

Figure 2.6: The Q-Learning algorithm (taken from [98]).

the estimate for the current state by making use of the existing estimate for the next state.

The two most common Temporal-Difference Learning methods are *Sarsa* and *Q-Learning*.

2.11.1.1 Sarsa

In the *Sarsa* algorithm an estimate of the action-value function is maintained for every state-action pair. After each learning step the resulting reward r_{t+1} and next state s_{t+1} is observed and the estimate for the current state-action pair $Q(s_t, a_t)$ is corrected towards the value resulting from the current reward and the discounted estimate of the next state-action pair $Q(s_{t+1}, a_{t+1})$ with a step-size parameter α:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right].$$

As the algorithm continually estimates Q^π with respect to the same policy π which is also used for generating the learning steps it is a so-called on-policy algorithm [94, 96, 98].

Figure 2.5 gives a sketch of the algorithm.

2.11.1.2 Q-Learning

If the next state-action pair used in the update is not determined according to the current policy π but instead by selecting the best possible action the estimated action-value function Q directly approximates the optimal action-value function Q^*:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

This algorithm is called *Q-Learning*. It is an off-policy algorithm since the learned action-value function is independent of the policy being followed [98, 103]. The basic algorithm is given in Figure 2.6.

3 Requirements and Design Goals

This work focuses on complex behaviors for cooperative multi-agent applications which pose a challenging task in highly dynamic environments as they are encountered in many real-world applications. Pragmatic and efficient methods are required for programming agent behaviors that are able to cope with necessary real-time requirements, only partial or noisy observability of the environment, and the unpredictability of dynamic environments [37]. These methods also have to be able to scale well to very large and complex systems.

Traditional AI approaches, that for instance try to generate the plans and actions of a team of robots through problem solving, usually fail to fulfill such requirements.

In the context of this work *pragmatic* methods are methods which strongly support the development of agent behaviors for complex applications. Developers should be provided with an adequate method for generating this kind of behaviors with as little effort as possible.

In the author's opinion the most important requirements for agent behavior programming architectures that are suitable for realizing complex real-world multi-agent tasks are the following:

- Modularity: Only when a large system is composed of smaller modules does the complexity of the whole system stay manageable. A modular decomposition promotes code reusability. Smaller building blocks which might be implemented and tested beforehand can be reused in possibly different contexts in order to construct more complex behaviors.

- Portability: The behavior architecture should be independent as far as possible of the robotic platform it is running on. Ideally a behavior architecture will be applicable to any robot system and will be portable to arbitrary software architectures. Also the application domain should have no influence on the selection of the behavior architecture.

  Many of the approaches presented in the previous chapter only support a small number of specific robot architectures or are tightly coupled to specific software environments.

- Versatility: Another requirement is that it should be possible to apply different styles of behavior programming. Behaviors might either be reactive or deliberative. There should be support not only for discrete but also for continuous behavior aspects. Behaviors might be required to engage in cooperation with other agents in multi-agent setups. Hand tuned-behaviors might be combined with optimized or machine learning behaviors. Modules of different styles of behaviors should integrate smoothly into one complex behavior. It should be possible to execute multiple behaviors concurrently.

Table 3.1: Comparison of some properties of different behavior architectures. (See Chapter 2 for details on the compared behavior architectures.)

	modular behavior decomposition	applicable for any platform	support for machine learning	direct support for continuous behaviors	concurrent behavior execution	support for multi-agent cooperations	customized programming language	availability of tools for programming	availability of tools for debugging
Behavior Language	yes	no	no	no	yes	no	yes	no	no
CDL / MissionLab	yes	no	yes	yes	yes	no	yes	yes	yes
COLBERT	yes	no	no	no	yes	no	yes	no	yes
Petri Net Plans	yes	yes	no	no	yes	yes	no	yes	no
HAL	yes	yes	no	yes	yes	yes	yes	no	no
DPA	yes	yes	no	no	no	no	no	no	no

- Usability: Specific behavior programming languages can support the rapid development of complex agent behaviors. Programming languages should be easy to understand and learn in order to reduce the time required for new developers to get familiar with the system. Also tools that support development and debugging of behaviors simplify the process of creating efficient behaviors.

Table 3.1 shows a comparison of properties of some of the behavior architectures presented in Chapter 2 with respect to the above requirements.

The XABSL architecture as it will be presented in the next chapter fulfills several of the aforementioned requirements very well and has proven to be suitable for complex real-world applications at least in the robotic soccer domain. In the next chapter the latest version of XABSL will be presented which is based on a significantly extended architecture and has been improved with respect to these design goals. In particular the versatility has been enhanced as concurrent behavior execution and continuous behavior aspects can be integrated more easily and also because it was investigated how machine learning concepts can be combined with hierarchical finite state machines. A new customized programming language makes the development of behaviors easier.

Although robotic soccer has been used as the primary test bed for the behavior architecture, it can be applied to a large variety of different applications.

4 Hierarchical Finite State Machines

As shown in the previous section none of the examined behavior architectures is able to satisfy all of the desired design goals. The main focus of this work is to develop an architectural concept and a behavior programming tool which complies with the design goals and all of the requirements stated in the previous section.

It was decided to use the behavior architecture of XABSL, which uses hierarchical finite state machines, as a starting point for developing the sought after behavior architecture. This decision was not only based on the fact that the first version of XABSL was already applied in the *GermanTeam* (cf. Section 1.2) to program the behavior of autonomous soccer playing robots. Hierarchical state machines have proven to be very well suited for the modular programming of robot behaviors, as has been shown by the successful application for different robots and leagues at *RoboCup* competitions for several years.

Another advantage of hierarchical state machines is that they provide a profound formal approach which has already been researched extensively. For instance, hierarchical state machines allow for formal analysis such as deciding reachability of certain states or model checking [3]. This might prove a considerable benefit especially when considering very large and complex behaviors where results which are available through formal analysis cannot easily be obtained manually.

Thus, it was not required to create a new behavior architecture from scratch. Furthermore, none of the other available behavior programming architectures were as well suited as XABSL to implement the decision making module required for robot soccer. Therefore, it was decided that the team kept using XABSL throughout the years following the introduction of XABSL in 2002 until the last tournament in 2008. We designed and integrated various improvements into the architecture in order to allow to program robot behavior even more efficiently. Many of the changes described in this section are based on experiences gained from the application in *RoboCup*.

Nevertheless, creating an architecture solely suited for realizing a robot soccer application is not the focus of this work. While robot soccer has always been the primary test application, XABSL does not have any application specific features or limitations. Therefore, by improving the hierarchical state machine behavior architecture in order to support common design goals towards cooperative real robot multi-agent applications, progress is not restricted to a specific application. See Section 6 for examples of application domains outside of robot soccer.

Some of the main improvements to the behavior architecture are concerned with the integration of alternative methods of behavior control which previously were impossible or difficult to apply in the original architecture of XABSL. Examples include concurrent behavior execution, continuous behavior methods, and cooperative multi-robot behaviors.

The usability has been increased by switching to a newly developed programming language which replaces specifying behaviors in XML. New tools have been created which support the development of behavior specifications.

The XABSL architecture, above mentioned improvements, new concepts, and features are described in the remainder of this section.

4.1 XABSL (2004)

The initial version of the *Extensible Behavior Specification Language* (XABSL) was developed by Martin Lötzsch et al. [65, 67] and was first applied by the *GermanTeam* in *RoboCup* 2002 [85]. The author of this work was a co-developer and has since been extending and maintaining the project. *XABSL* is a pragmatic approach for engineering agent behavior based on hierarchical finite state machines independent of the agent platform and architecture.

It consists of three main parts:

- The first is an agent behavior architecture based on hierarchical finite state machines.
- The second is the behavior specification language which, in the first version, was an XML dialect.
- The third is an execution engine, a class library which can execute the state machines directly on a target platform. This is achieved by interpreting an intermediate code which is generated automatically from XML source files containing the *XABSL* behavior description. The *XABSL* system also contains various tools, e.g. for documentation and debugging.

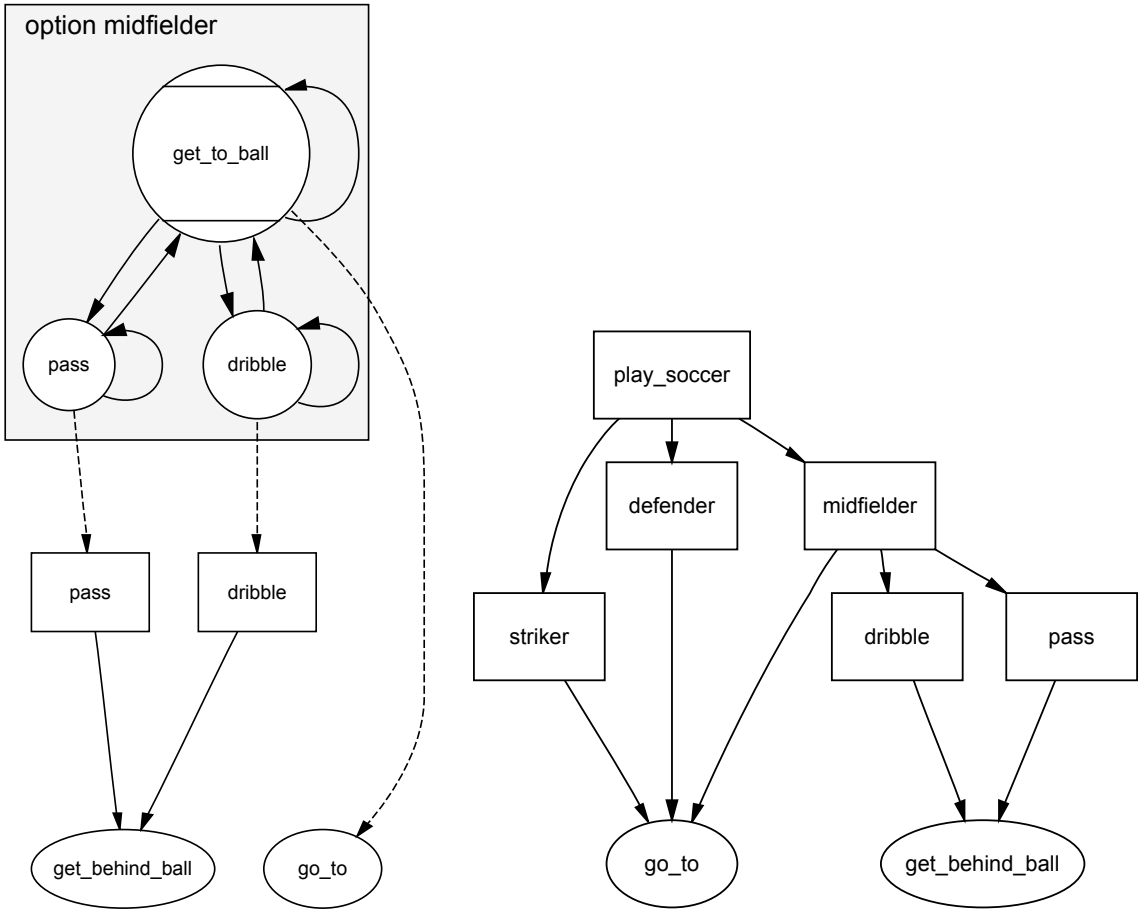
One major improvement besides architectural concepts that was introduced in later versions by the author was the replacement of the XML representation by a behavior programming language which has a more compact syntax. Programming directly in XML was very tedious and time consuming. The XML representation is still available as an automatically generated intermediate representation which is used e.g. for automatic documentation generation.

4.1.1 Hierarchical Finite State Machines

In *XABSL* hierarchical finite state machines are applied in order to model the decision making of an agent. A behavior specification defines a set of finite state machines, called *options* and a set of predefined behavior routines, called *basic behaviors*.

Options and basic behaviors are ordered in a hierarchy, where more complex options are composed of less complex options and basic behaviors. The hierarchy can be described as a directed, acyclic graph, called *option graph*. Each vertex in the option graph is either an option or a basic behavior. Basic behaviors are sinks in the option graph.

An option graph can define multiple agents, which can share options and basic behaviors. An agent is a rooted subgraph of the option graph which is spanned by a specific option, the *root option* of the agent. When executing the hierarchical state machine, the current state is defined as the subset of activated options along a directed path in the option graph starting from the *root option* and their respective states. The path of active option is called the *activation path*.



(a) An example option. The option *midfielder* consists of the three states *get_to_ball*, *pass*, and *dribble*. The initial state *get_to_ball* is marked with two horizontal lines. The dashed lines indicate the subsequent options for each of the states.

(b) An example option graph. It shows the decomposition of the root option *play_soccer* into the options *striker*, *defender*, *midfielder*, *dribble*, and *pass* and the two basic behaviors *go_to* and *get_behind_ball*.

Figure 4.1: An XABSL example from a robot soccer scenario.

It lies in the responsibility of the execution engine to check whether the option graph is acyclic. While it is possible to specify behavior which contains loops this would not result in meaningful behavior. Thus, the execution engine has to check at runtime whether the option graph contains loops.

Figure 4.1(b) shows an example of an option graph.

4.1.1.1 Options

As mentioned above each option is a state machine. Therefore an option consists of a finite set of states. Each state definition consists of two parts. One is the *decision tree* which defines transitions to other states of the option. The other is an *action definition*, which specifies a subsequent option or basic behavior which should be executed as long as the

state is activated. Furthermore, action definitions can specify output symbol assignments, which allow the setting of certain output variables while the state is active.

The options and basic behaviors defined in the action definitions of an option define the outgoing edges of the option in the option graph, since these options and basic behaviors are called from the different states of the option.

In the first version of XABSL each state had to define exactly one option or basic behavior to be executed while the state is active. In later versions developed in this thesis the expressiveness of hierarchical state machines was increased largely by introducing concurrency. Now an action definition can define any number of options or basic behaviors, which then are executed concurrently while the state is active. The current state during execution is then no longer represented by a path but rather a directed tree – the *activation tree*. The tree has the root option as its root node while basic behaviors are optional leafs of the tree.

One of the states of each option is marked as the *initial state*. The initial state is assumed when an option becomes active.

Any state of the option can be marked as a target state. A calling option can query whether a subsequent option has reached one of its target states. This can be applied for instance to notify that a certain task of an option has been accomplished.

Figure 4.1(a) shows an example of an option.

4.1.2 Interfacing the Agent

The *XABSL* behavior is always only a part of an agent software architecture. Depending on the given application the surrounding software e.g. might be responsible for processing sensor inputs, creating a model of the environment, managing communication, and motion generation, while the *XABSL* behavior is responsible for high-level decision making. The behavior interacts with the surrounding software through symbols. The behavior specification defines a number of *Input* and *Output Symbols* which represent the interface through which the behavior communicates with the surrounding software. *Input Symbols* can be e.g. sensor inputs, world model data, or messages received from other agents, while *Output Symbols* can be e.g. motor controls or messages to be sent to other agents.

4.1.3 XML Description Dialect

The hierarchical state machines defined in *XABSL* are specified in XML. In the initial version there is no custom programming language. Instead, state machines are programmed in XML code. This enables the use of standard XML techniques such as XSLT processors for validation and compilation. Figure 4.2 shows an example of the XML code.

4.2 Concurrent Behavior Execution

One of the major improvements to XABSL hierarchical state machines is the introduction of concurrent behavior execution. In the original version in each state exactly one subsequent action had to be selected. In each execution exactly one basic behavior was selected and executed. This leads to option activation paths consisting of a number of options and

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dummy-doc-type [
 <!ENTITY my-symbols SYSTEM "../my-symbols.xml">
 <!ENTITY my-basic-behaviors SYSTEM "../my-basic-behaviors.xml">
 <!ENTITY options SYSTEM "../options.xml">
]>
<option xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
xmlns:xi="http://www.w3.org/2001/XInclude"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
../../../Xabsl2/xabsl-2.1/xabsl-2.1.option.xsd" name="striker"
initial-state="initial">
 &my-symbols;
 &my-basic-behaviors;
 &options;
 <state name="initial">
  <subsequent-basic-behavior ref="go-to">
   <set-parameter ref="go-to.x">
    <minus>
     <decimal-input-symbol-ref ref="ball.x"/>
     <decimal-value value="8"/>
    </minus>
   </set-parameter>
   <set-parameter ref="go-to.y">
    <decimal-value value="11"/>
   </set-parameter>
  </subsequent-basic-behavior>
  <decision-tree>
   <transition-to-state ref="initial"/>
  </decision-tree>
 </state>
</option>
```

Figure 4.2: An example of an XABSL option in XML code.

one active basic behavior. This limitation has been removed. States can reference any number of actions allowing multiple actions to be executed in parallel. States also do not need to reference any action at all, which allows having options as leafs in the activation tree. Basic behaviors are no longer mandatory as the last element in a chain of actions, but rather they are optional leafs of the activation tree (cf. 4.1.1).

Allowing concurrency greatly increases the versatility of the architecture. Different independent parts of the behavior can be executed concurrently. Having the possibility for concurrent execution can be a necessary requirement for certain applications that contain subtasks which need to be run in parallel independently from each other. Even if it is possible to program a certain behavior without using concurrent execution, being able to do so might allow drastically simplified solutions.

In the literature one can also find hierarchical finite state machines which allow concurrency, for instance, in UML statecharts [44]. All of the other architectures for behavior programming presented in Section 2 also include some form of support for concurrent behaviors (compare with Table 3.1 in the previous section).

Applications implemented in earlier XABSL versions had to be structured in a way where the main task of decision making is to select exactly one current basic behavior and necessary parameter values out of a set of mutually exclusive basic behaviors. With concurrent actions it is no longer necessary to focus on the selection of one basic behavior. Instead, output symbols can also be used as the main method of providing results from

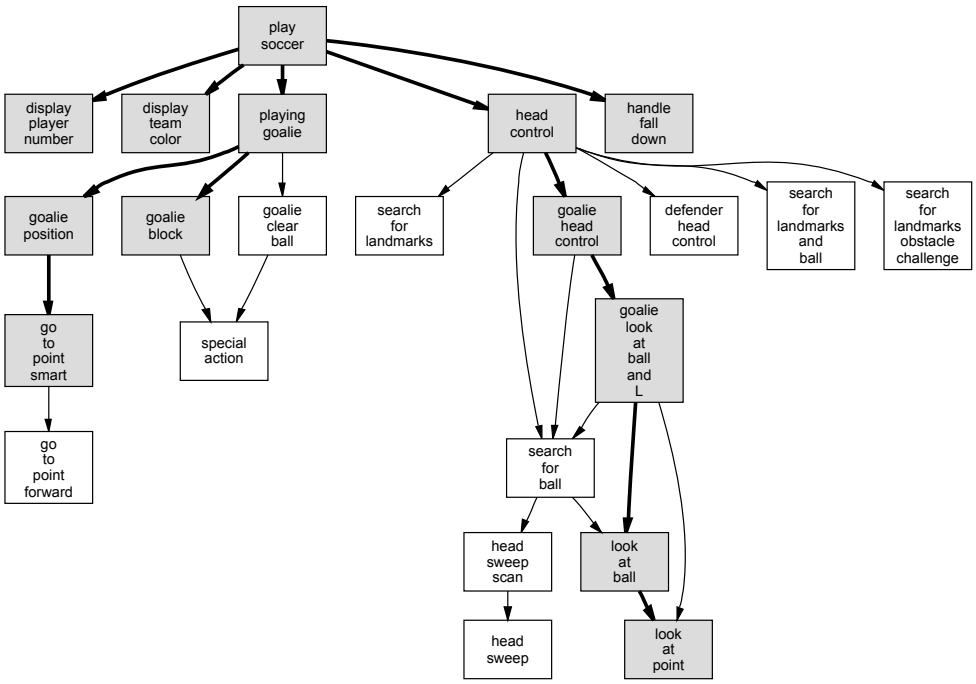


Figure 4.3: An example option graph from a robot soccer goalie behavior which makes use of concurrent option execution. Boxes represent options. The edges show which options might get activated from another option. The highlighted options, basic behaviors and edges show an example of a possible option activation tree, showing which behaviors are activated during one specific time step.

the hierarchical state machine. In this case basic behaviors can be used for executing specific behavior routines implemented elsewhere.

Adding concurrent execution also adds the possibility of producing conflicting actions by setting the same output symbol from different concurrently executed options. Such conflicts are solved by applying a strict prioritization of actions. In each execution cycle the XABSL execution engine will perform a depth-first search through the active option activation tree. Actions encountered first will be executed first, thus having a lower priority than the actions following them, since their outputs can still be overwritten afterwards. It is in the responsibility of the application programmer to avoid undesirable resulting behaviors due to conflicting outputs from concurrently executed behaviors. Furthermore it is not allowed that the same option or basic behavior gets activated more than once by the active options. Therefore, the active option activation tree forms a subtree of the option graph, as each option or basic behavior can be activated only through exactly one path starting at the root option of the current agent.

Figure 4.3 shows an example of an option graph and an option activation tree from robot soccer. The root option *play_soccer* is executing five concurrent behavior options simultaneously. While the option *playing_goalie* contains the main soccer behavior, the other concurrent options contain reactive behaviors that should be active independent from the current state of the behavior (in option *handle_fall_down* for handling situations when the robot has fallen over) or implement debug displays (options *display_player_number* and *display_team_color*). The option *head_control* controls the gaze direction of the robot

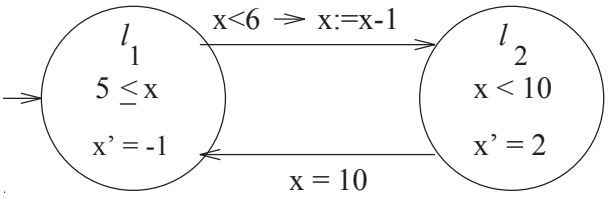


Figure 4.4: An example of a hybrid automaton (taken from [2]).

which can be controlled independently from the leg motion of the robot. Furthermore, the options *goalie_position* and *goalie_block* are executed in parallel, allowing the robot to try to catch a ball rolling towards it, while at the same time positioning itself.

4.3 Integration of Continuous Behavior Control

With finite state machines only the discrete aspects of behavior control can be modeled. Continuous behavior aspects such as controlling real-valued output variables cannot be represented through discrete state transitions.

One solution for modeling mixed discrete-continuous dynamical systems or behavior control, which includes discrete and real-valued state variables, is the formalism of hybrid automata [2]. A hybrid automaton is a finite state machine which is augmented with continuous state variables. Continuous dynamics can be modeled through differential equations in flow conditions and invariants. Applications of hybrid automata are usually concentrated on formal analysis, model-checking, and verification. Formal methods can be applied in order to verify certain properties of hybrid automata. An architecture for behavior control which is based on hybrid automata can be found in [35, 78] (cf. Section 2.9). A small example of an automaton consisting of two discrete states and one continuous variable is shown in Figure 4.4. In a hybrid automaton each discrete state or *location* is labeled with *invariants* which define constraints the continuous variables must hold while the state is selected. States can also be labeled with *activities* which define differential equations that describe how the continuous variables change over time. Transitions between locations can be labeled with *guard conditions* which the continuous variables must fulfill when the transition is taken and *assignments* that describe instantaneous changes to the continuous variables that occur when a transition is taken.

Other behavior control methods that are able to generate continuous outputs include methods for geometric path planning, for instance through the use of potential fields. Such methods usually provide the lowest level of behavior control and lend themselves to being included at the bottom of a hierarchical behavior control module.

Often the output of continuous behavior modules can be combined, for instance through a weighted vector summation, in order to assemble complex behaviors. This also is one of the approaches for creating assembled behaviors in the *Configuration Description Language* [70] (cf. Section 2.5). An example is shown in Figure 4.5.

The XABSL architecture originally was mainly concerned with discrete behavior aspects, such as selecting the currently active basic behavior. Since enumerated output symbols were the only available type of output mechanism besides basic behaviors, gener-

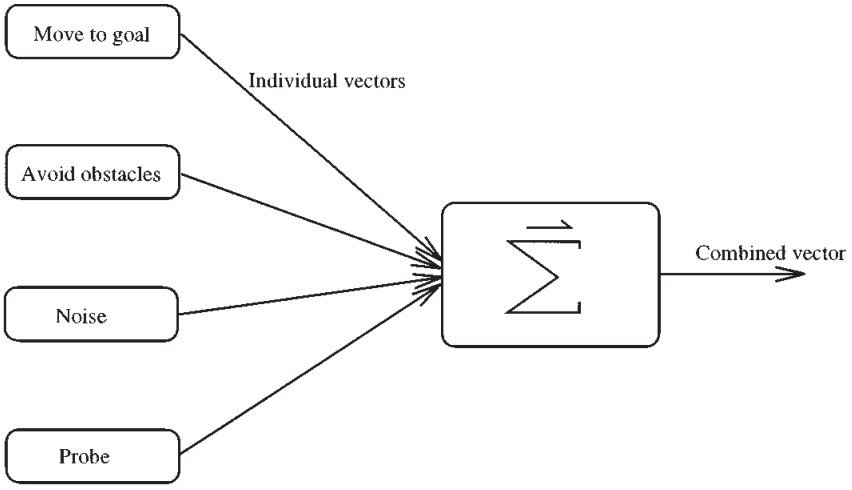


Figure 4.5: An example for vector summation of continuous behaviors (taken from [70]).

ating continuous output values directly from the state machine was not possible. Continuous behaviors such as potential field based methods had to be realized as basic behavior.

Our main improvement to the hierarchical state machines in XABSL which is necessary to support continuous behaviors is the introduction of decimal output symbols. Decimal output symbols are continuous variables that can be written from inside of the hierarchical state machine. With decimal output symbols it is possible to emulate some of the features of hybrid automata. Also, if the input from primitive continuous behaviors is available as input symbols, it is possible to realize vector summation as described above. For the sake of completeness boolean output symbols have also been introduced.

Another new feature in XABSL are internal symbols. These symbols are only available inside of the hierarchical state machines, but have no equivalent in the surrounding software environment. Decimal internal symbols are also supporting continuous behaviors as they can be used to store continuous state variables.

Figure 4.6 shows how the hybrid automaton from Figure 4.4 can be translated into an XABSL state machine. The continuous state variable of the hybrid automaton can easily be reproduced using an internal decimal symbol. If access to the continuous variable is required outside of the state machine, a decimal output symbol could be used as well. The translation is not exact, as there have to be certain differences between the hybrid automaton and the state machine. Hybrid automata are non-deterministic models of the possible dynamics of a hybrid system. On the other hand, state machines used for behavior control need to be deterministic descriptions of the behavior of an agent. Therefore, when translating hybrid automata it is assumed that a transition is taken as soon as the guard condition holds. Invariants are not required and, thus, are ignored. Another difference is that XABSL state machines do not provide assignments at transitions. Assignments can be emulated with an additional state which is only active once and will execute the assignment as a state action. Likewise, describing the dynamics of the continuous variables with differential equations in activities is not supported in XABSL. Since the XABSL state machine will be executed in discrete time steps, the integration of the differential equations can only be approximated stepwise. If a fixed known execution cycle time is assumed this can be done straightforward.

```
internal float x;

option hybrid_example {
  initial state l1 {
    decision {
      if (x < 6)
        goto l1_l2;
      else
        stay;
    }
    action {
      x = x - 1;
    }
  }

  state l1_l2 {
    decision {
      goto l2;
    }
    action {
      x = x - 1;
    }
  }

  state l2 {
    decision {
      if (x == 10)
        goto l1;
      else
        stay;
    }
    action {
      x = x + 2;
    }
  }
}
```

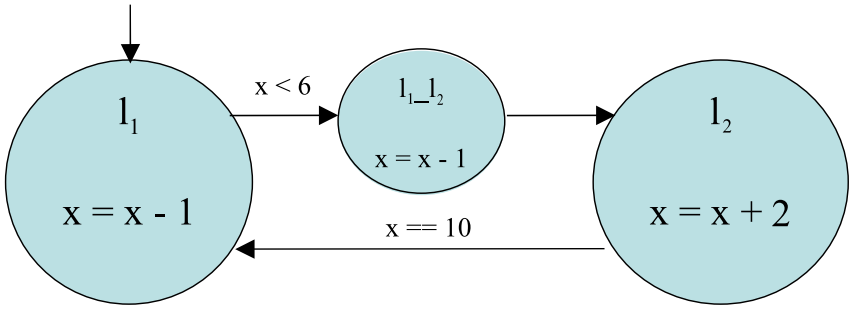


Figure 4.6: A deterministic XABSL translation of the non-deterministic hybrid automaton from Figure 4.4.

4.4 Cooperative Multi-Robot Systems

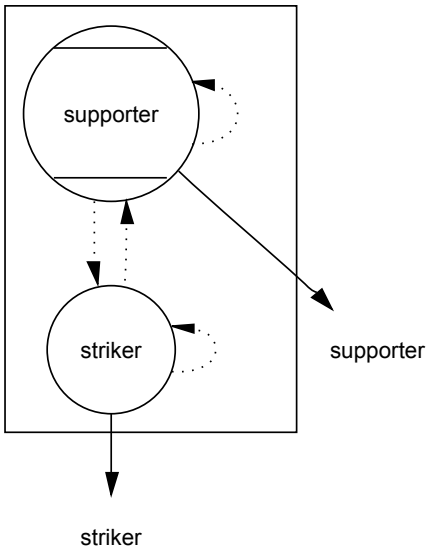
In previous versions the XABSL architecture did not support cooperation between agents directly. Instead programmers had to implement cooperation between agents externally and provide its result to the hierarchical state machine. For example, in a scenario where a team of autonomous robots should carry out a given set of tasks, the task assignment result – in that case the current task assigned to each robot – would be considered an input variable.

Since task assignment usually is state based, it can conveniently be implemented with hierarchical state machines. XABSL provides the following features to support cooperation between multiple communicating agents [83]:

- A typical requirement is that a certain state of a state machine can only be executed by at most a given number of agents at the same time. The maximum number of agents that can execute a state is called the capacity of the state. A possible example might be a team of robots navigating through a narrow passage which can only be entered by a certain number of robots at once without blocking each other. Another example and its implementation in XABSL is shown in Fig. 4.7(a).

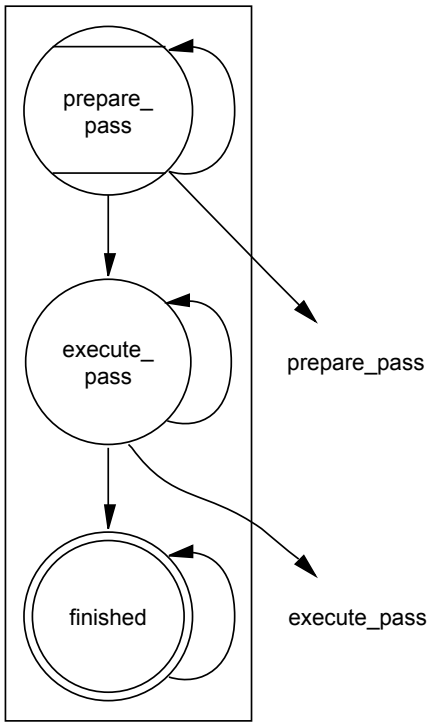
- Another requirement is that the actions of multiple agents might need to be synchronized. This can be realized by specifying, that all agents currently executing

```
option play
{
  common decision
  {
    if (ball.distance <
        teammate.ball.distance)
      goto striker;
    else if (true)
      goto supporter;
  }
  initial state supporter
  {
    action
    {
      supporter();
    }
  }
  state striker capacity 1
  {
    action
    {
      striker();
    }
  }
}
```



(a) State *striker* has capacity of one

```
option pass
{
  bool @receiver_or_sender;
  initial state prepare_pass
  {
    decision
    {
      if (action_done)
        goto execute_pass;
      else
        stay;
    }
    action
    {
      prepare_pass(is_sender = @is_sender);
    }
  }
  state execute_pass synchronized 2
  {
    decision
    {
      if (action_done)
        goto finished;
      else
        stay;
    }
    action
    {
      execute_pass(is_sender = @is_sender);
    }
  }
  target state finished
  { action {} }
}
```



(b) State *execute_pass* is executed synchronized

Figure 4.7: Two examples from robot soccer: Example a) shows a state machine for role assignment. Only one of the field players shall attack the ball. Therefore the state *striker* has a capacity of one. Example b) shows an option for pass play. Only after both robots are finished preparing for the pass, will they enter the state *execute_pass* synchronously.

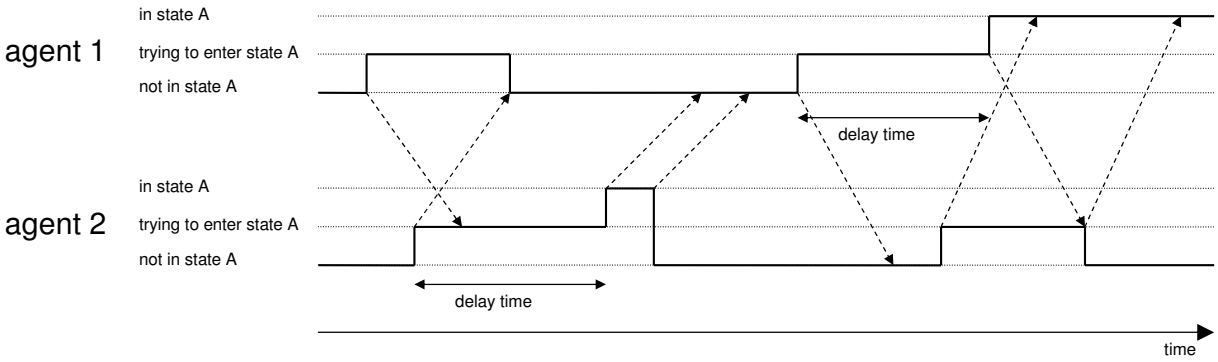


Figure 4.8: An example of the coordinated execution of a state with a capacity of one by two agents. Agent 2 has higher priority than agent 1. The dashed arrows depict the amount of time required for signaling a state change to the other agent. The delay time before entering the state is chosen as twice the amount of time required for transferring a message from one agent to another. Therefore, there are no capacity conflicts.

an option are required to enter a certain state of the option at the same time. If an agent tries to enter the state it will wait until every other agent is also ready to enter the state. An XABSL example is shown in Fig. 4.7(b). Optionally a minimum number of agents that are required to enter the synchronized state can be specified. The set of agents which execute an option synchronously can also be a subset of all the agents of a cooperative scenario. Only the agents that are currently executing the behavior option which requires synchronization are taken into consideration.

These two features allow the programmer to specify most common cooperation tasks using comfortable and comprehensible methods. More complex cooperation tasks with specific communication requirements might not be realized in the state machine directly. If access to incoming and outgoing messages is provided through symbols, however they can be integrated easily.

In a typical implementation for realizing complex multi-robot applications with XABSL there is a single hierarchical behavior which is executed by all of the robots. The behavior can contain state machines for task allocation in the top levels of the hierarchy. Applications are not limited to homogeneous robot teams. In case of heterogeneous multi-robot applications, the different capabilities and limitations of each robot can be made available to the behavior through input symbols. According to these symbols robot specific sub-behaviors can be selected.

In most multi-agent environments, e.g. in every real multi-robot application, one cannot assume that messages between agents will be sent and received instantaneously. Therefore, conflicts may arise, e.g. when two agents try to enter a state with a capacity of one at nearly the same time. In order to prevent such conflicts, some form of negotiation is necessary. In the proposed extension of XABSL the following negotiation pattern is applied: Whenever an agent tries to enter a state with a capacity it signals this to other agents and waits for a certain amount of time before entering the state. If the number of agents trying to enter exceeds the available capacity of a state, a user-defined agent prioritization is applied. It is easy to see that increasing this delay leads to an increased protection against capacity conflicts. Only if the delay time is greater or equal

to the maximum round trip time of sending a message to all other agents and receiving respective responses, is it guaranteed that the number of agents executing a state will never exceed the capacity of the state (cf. Fig. 4.8). On the other hand increasing the delay time leads to a reduced reactivity of the state machines. Thus, there is a trade-off between prevention of possible conflicts and reactivity. In some applications it might be critical to guarantee that the capacity of a state never gets exceeded, not even for very small amounts of time. In other applications it might be more important that decisions are made as quickly as possible (e.g. in the robot soccer scenario). Therefore the delay time is a parameter selectable by the application programmer.

Examples which show that multi-robot applications can be realized easily using these features are described in Section 6.1.3.

4.5 Machine Learning and Optimization

On the one hand decision-making methods which do not require the developer to specify in detail how an agent reacts in every imaginable situation, such as machine learning algorithms or optimization methods, can alleviate the development, and, in many applications where finding the optimal behavior is not trivial, might outperform hand coded behavior. On the other hand these methods are often not able to cope with the complexity of large real-world tasks especially when faced with noisy sensor data, unpredictable dynamics, or uncertain actions. Furthermore, there are often situations where the developer wants to specify the exact behavior of an agent directly, for instance, when there is an obvious optimal strategy that can be implemented easily.

As has been stated in Section 2.11 applying machine learning or optimization approaches on parts of complex behaviors can be very useful as specific subtasks might be especially suited for the application of such methods. The decomposition of behavior specified with hierarchical finite state machines supports combining different styles of behavior programming very well. Learning or optimization can be used on one of the hierarchy levels, while hand-coded behavior options can be used on others, for instance, as primitive behaviors to be selected or parameterized by the learned or optimized behaviors. Learning methods can also be applied on multiple hierarchy layers, where the resulting behaviors for one layer are used as the basis for learning the next layer. This leads to the layered learning paradigm as it is described by Stone and Veloso [97]. The key principles of layered learning are given in Table 4.1.

Table 4.1: The key principles of layered learning [97].

1.	A mapping directly from inputs to outputs is not tractably learnable.
2.	A bottom-up, hierarchical task decomposition is given.
3.	Machine learning exploits data to train and/or adapt. Learning occurs separately at each level.
4.	The output of learning in one layer feeds into the next layer.

In order to interface a learning or optimization algorithm with a hierarchical finite state machine, the interfacing facilities of XABSL can be applied readily. The result of a discrete or continuous external action selection mechanism can be provided through

input symbols. If calculations have to be performed at every step, for instance in order to update learned values, this can be realized using basic behaviors. Feedback to the learner or optimizer, for instance current state information or rewards, can be provided with output symbols.

Another possible application of optimization methods is the automatic tuning of certain parameters of an option. If a certain behavior option has been implemented but the optimal values for continuous parameters are unknown, optimization methods can be applied in order to determine the optimal parameter values. Parameter values can include threshold values for triggering transitions or parameter values for referenced options or basic behavior.

For instance, we determined the optimal parameters of the behavior of grasping the ball semi-automatically using an optimization approach based on *Asynchronous Parallel Pattern Search* [42, 57]. The aim is to secure the ball under the chin of the robot quickly and reliably. See Section 6.1.4 for details on the results.

In the XABSL architecture, besides providing the current parameter values to the option through input symbols, another possibility is to specify the parameters to be optimized as option parameters. When optimizing the option the current parameter values can be set using the debugging interface which is also used to manually execute the option. After optimal parameter values have been found these can be set from the calling option, thus there is only one place in the behavior where the actually employed parameter values are specified.

The above example of optimizing the ball grasping behavior has been investigated by several teams in the *RoboCup Four-Legged-League*. Different optimization and learning methods have been applied. For example, besides the above mentioned optimization algorithm, policy gradient learning [27, 28], Sarsa reinforcement learning [55], and evolutionary methods [45] have been applied successfully improving some aspects of a ball grasping behavior for the *Sony Aibo* robot. This is a case study which documents that a small subtask of a complex task can be very well suited for machine learning methods, though this is not possible when considering the complete task, in this case playing autonomous robot soccer in the *RoboCup Four-Legged-League*. Therefore, having a hierarchical behavior architecture which supports machine learning of subtasks is desirable.

Independently from the behavior architecture, machine learning and optimization algorithms require evaluation functions which provide a method of determining the current performance of an agent's behavior. Finding a good evaluation method is often crucial. When working with physical robots evaluation function values will in most cases be noisy. Evaluation function values can either be determined directly by means of the agent or be provided by an external instance, which is not available in the normal operation of the agent. In the latter case obtained values might be more exact whereas applications are possibly limited as additional infrastructure is required. For instance, when optimizing the walking speed of a robot, it may be necessary to use an external localization method, such as a laser range finder or a ceiling camera system, in order to measure the exact walking speed of the robot with the accuracy required for reliably comparing walking gaits.

In the above example of grasp learning, it can easily be evaluated whether the task was accomplished or not. Sensors of the robot can be employed in order to determine whether the ball was grasped successfully. Thus, the most obvious evaluation value is a binary

signal. When each test run consists of several trials the result can be averaged leading to a discrete integer value specifying the number of successful trials. Having only a discrete evaluation value can be a drawback for many optimizations methods. Thus, it might be advantageous to generate continuous reward signals, in this case, by using a different optimization criterion, for example the time until the ball was grasped successfully can be regarded, when not only grasping success but also speed is to be optimized.

5 Behavior Specification with Extended Hierarchical State Machines

In this section the resulting architecture is described which is based on the architecture from Section 4.1 including the extensions presented in the previous section. Also design motivations are presented, which explain why certain design or implementation decisions were met.

5.1 Concurrent Hierarchical Finite State Machines

In XABSL the behavior of an agent is modeled with concurrent hierarchical finite state machines. The hierarchical state machine is composed out of behavior components, called *options*, which contain simple finite state machines.

5.1.1 Option graph

An option can be seen as an abstraction of a certain behavior task and can be used as a high level primitive by other options. Thus, the set of options forms a hierarchy with primitive options, which do not rely on other options, on the lowest level, and options, which combine existing options into more complex behaviors, on top.

Options can also reference so called *basic behaviors*. Basic behaviors are primitive behaviors implemented elsewhere which can be used as additional behavior building blocks.

Options and basic behaviors can have parameters which are specified when the option or basic behavior is instantiated from another option.

The option hierarchy forms a directed acyclic graph, the *option graph*. Vertices of the option graph are options and basic behaviors. Sinks of the graph are either basic behaviors or primitive options. An agent is specified by defining one of the options, for instance a source of the option graph, as the *root option* of the agent. The agent then consists of the rooted subgraph of the option graph spanned from the root option. Figure 5.1 shows an example of an option graph.

During behavior execution, the options and basic behaviors that are activated at a specific time step form a rooted tree which is a subtree of the option graph, the so called *option activation tree*.

While it is possible that an option gets activated via different paths in the option graph (e.g. in the example in Figure 5.1 the option *go to point* can be referenced from options *striker* or *supporter*), it is not allowed that an option is activated via different paths at the same time. This is necessary as the semantics for that case would be ambiguous. It would be unclear whether there should be multiple instances of a behavior. If there is only one instance of each behavior then parameterizations would become undefined. Therefore,

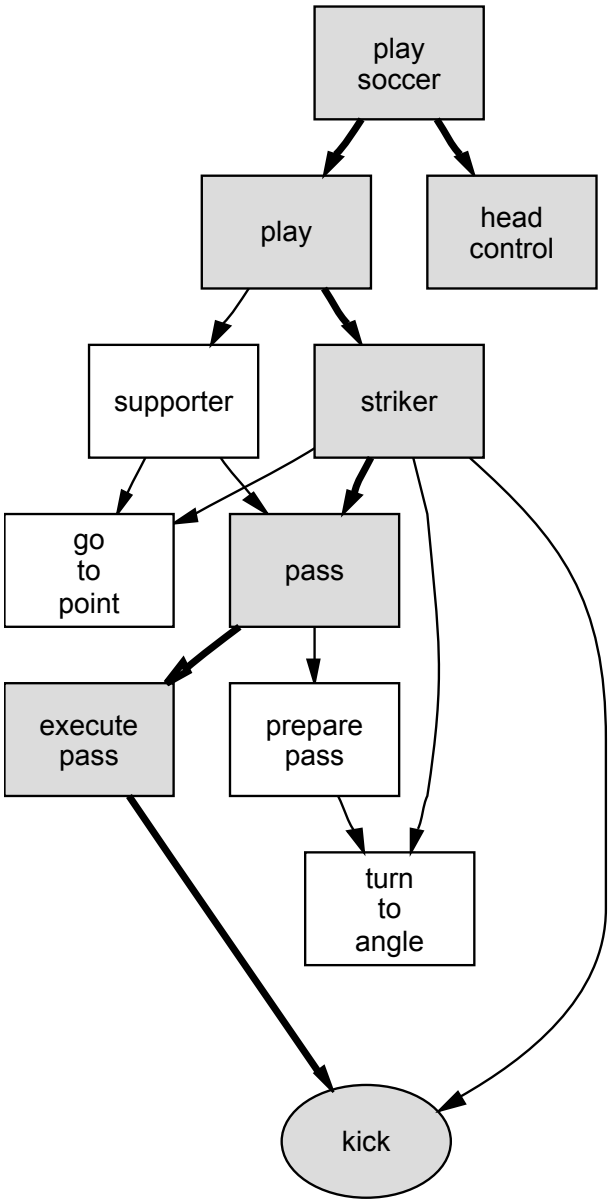


Figure 5.1: An example option graph from a simple two-player robot soccer behavior. It shows the decomposition of the task of playing soccer into several options and a basic behavior. Boxes represent options, ellipses represent basic behaviors. The edges show which options might get activated from another option. The highlighted options, basic behaviors and edges show an example of a possible option activation tree, showing which behaviors are activated during one specific time step.

this kind of multi path activation is not allowed, and thus, the subgraph of activated options is always a rooted tree. It lies in the responsibility of the execution engine to check whether this property holds and whether the option graph is acyclic. Violations will result in a runtime error when initializing the engine.

5.1.2 State Machines

Each option is a finite state machine. Each of the states of an option defines an ordered list of actions that are executed while the state is active. Figure 5.2(a) gives an example showing the states of an option. As described above, the actions of a state can be references to other options or basic behaviors. Actions can also be output symbol assignments, setting output variables of the agent behavior (e.g. motor speeds, etc.). All actions of a state are executed concurrently as long as the state is active. This can lead to conflicts, for instance when concurrently active options are trying to write to the same output symbols. Therefore, the ordering of the actions is used to define an unambiguous precedence among concurrent actions, defining the results of which actions might eventually overwrite results of other active actions.

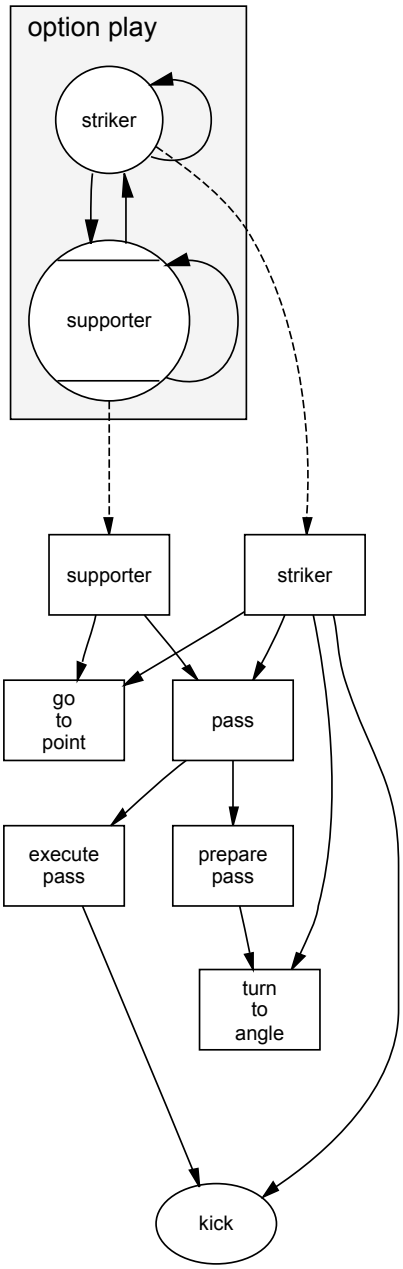
In order to define transitions between states *decision trees* are given, which are evaluated at each execution time step and determine the next state of an option. Figure 5.2(b) shows an example of the decision of one of the states of an option. The decisions can be based on different information such as values from option parameters passed from the calling option, timing information such as how long the current option or state is activated, or input variable values which can contain values from the agent's world state.

One of the states of each option is marked as the *initial state*. The initial state is activated whenever an option was not active in the previous execution cycle. This means that the state of an option is not reset to the initial state if the activation path changes. Even when the calling option changes, the option will remain in its current state. Only if the option was not activated for one or more execution cycles, will its state be reset to the initial state.

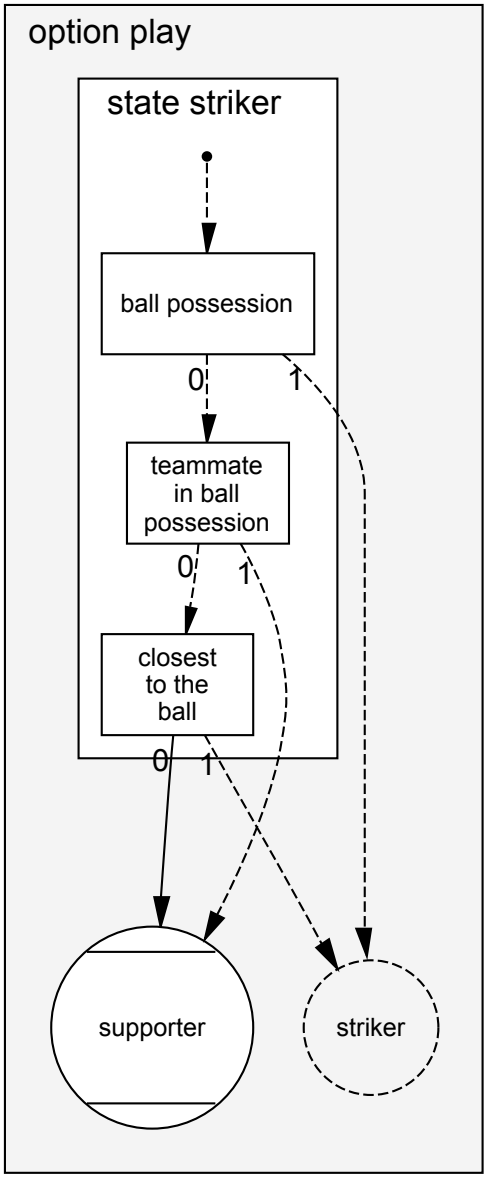
There is one specific built-in feature for allowing an option to give feedback to its caller. Since options are often called in order to perform a specific task and need to be active until this task has been carried out, the caller should be able to query whether the called option has finished its operation. This is supported through a simple mechanism, which allows any of the states to be marked as a *target state*. The calling option can query whether an option has reached one of its target states. If more elaborate feedback is required such as return values or a feedback on whether the task of the option was performed successfully or not, other means have to be applied.

5.1.3 Interaction with the State Machine

The XABSL hierarchical state machine is always running inside of a software agent system. The XABSL execution engine will be called at regular intervals, e.g. whenever new processed sensor data is available or at fixed timed intervals. According to its current state and inputs the hierarchical state machine will then determine its next state and output. Any preprocessing required for decision-making must therefore be completed in advance. Depending on the particular application preprocessing could include but is not



(a) One of the options of the option graph from Figure 5.1 is shown with its states. The diagram shows possible transitions between the state of the option, as well as which options are activated while each of the states is active and the corresponding subgraph of the option graph, which shows which options might get activated in consequence. The initial state of the option is marked by two horizontal lines.



(b) The decision tree of one of the two states of the option. The circles at the bottom of the graph represent state transitions. The dashed circle represents a situation where the state machine does not change its current state.

Figure 5.2: Examples of an option and its states and a decision tree.

limited to processing sensor information, updating a world model, or processing other inputs such as communicated messages or user inputs.

Inputs are made available to the state machine through application-specific statically typed symbolic variables and parameterized functions, the so-called *input symbols*. Since one of the design goals was application and platform independence there are no predefined variables, for example for accessing the world model of a robot. Instead the behavior programmer has to select all the relevant input variables and functions. Input symbols and constants can be combined with arithmetic and logic operators in order to form expressions to describe conditions for decision trees or parameter and output assignments. Custom arithmetic functions (e.g. *"distance_to(x, y)"*) can easily be added as input symbols.

Analogously, *output symbols* have to be defined by the behavior developer for each output value generated by the behavior. Output values could be locomotion requests, joint positions, motor speeds or other values depending on the respective application.

Internal symbols are symbols which do not have an equivalent outside of the XABSL state machine. *Internal symbols* are useful for variables that do not have relevance outside of the state machine, for instance for continuous state variables or for data exchange between options.

Symbols can have one of the following types: *decimal* for real-valued continuous variables, *boolean* for binary variables, and *enumerated* for discrete user-defined enumerated variables. Type safety is guaranteed by the compiler.

5.1.4 Multi-Agent Cooperation Facilities

As described in Section 4.4 XABSL provides features which simplify programming multi-agent cooperative behavior. A state of an option can either have a specific capacity, and thus only be executed by a certain number of agents at the same time, or a state can require all agents to enter it synchronously. These two very simple features allow the realization of many different cooperative scenarios. In many cases no further effort is required in order to perform cooperative behavior tasks.

5.2 Description Language

Figure 5.3 shows an example of an option specified in the XABSL programming language.

There are language elements for options, their states, and their decision trees. Boolean logic (||, &&, !, ==, ! =, <, <=, >, and >=), simple arithmetic operators (+, −, ∗, /, and %), enumerations, and conditional expressions (a ? b : c) can be used for the specification of decision trees, parameters of subsequent behaviors, and values of output symbols. Custom arithmetic functions (e.g. *"distance_to(x, y)"*) that are not part of the language can be easily defined and used in instance documents.

Symbols are defined in XABSL instance documents to formalize the interaction with the software environment. Interaction means access to input functions and variables (e.g. from the world model) and to output functions (e.g. to set requests for other parts of the information processing). For each variable or function that one wants to use in certain conditions, a symbol has to be defined. This makes the XABSL framework independent from specific software environments and platforms. The developer may decide whether to

grab-ball-with-head.xabsl

```
/** Grabs the ball with the head */
option grab_ball_with_head {
  common decision {
    /** ball distance greater than 200 mm */
    if (ball.time_since_last_seen_consecutively < 200 &&
        ball.consecutively_seen_time > 100 &&
        ball.seen.distance > 200 &&
        ball.seen.distance < 800)
    {
      goto approach_ball;
    }
  }
  initial state approach_ball {
    decision {
      /** state running less than 300 ms */
      else if (state_time < 300) {
        stay;
      }
      /** grab possible */
      else if (ball.time_since_last_seen_consecutively < 300 &&
               ball.consecutively_seen_time > 100 &&
               ball.seen.distance < (ball.play_ball_precisely ? 90 : 180) &&
               ball.seen.angle < (ball.play_ball_precisely ? 15 : 20) &&
               ball.seen.angle > (ball.play_ball_precisely ? -15 : -20))
      {
        goto grab;
      }
      else {
        stay;
      }
    }
    action {
      head_control_mode = search_for_ball;
      approach_ball(look_at_ball_distance = 500,
                    slow_down_distance = (ball.play_ball_precisely ? 380 : 350),
                    slow_speed = (ball.play_ball_precisely ? 100 : 170));
    }
  }

  state grab {
```

Figure 5.3: Example XABSL source code for the option *"grab-ball-with-head"*. It starts with the definition of a common decision tree (a decision tree that applies to all states of the option) and then continues with the implementation of the state *"approach-ball"*. Here the source code is shown in the editor of Microsoft Visual Studio, for which an XABSL syntax highlighting and code completion plugin exists.

express complex conditions in XABSL by combining different input symbols with boolean and decimal operators or by implementing the condition as an analyzer function in C++ and referencing the function via a single input symbol.

An XABSL agent behavior implementation is distributed over many source files, which helps the behavior developers to keep an overview over larger agents and to work in parallel.

Before the programming language was introduced in 2005 (initially under the name YABSL or XTC) [68, 92] behavior programming had to be done directly in XML. The XML representation is now only used as an intermediate format which can easily be parsed automatically using standard tools. The validation of XML sources based on schemes is replaced by syntax checking performed by the XABSL compiler. The availability of a tailor-made programming language greatly simplified programming behaviors and therefore increased the usability of XABSL significantly. The programming language has the advantage of giving a much more concise representation, thus reducing the size of the source code by more than 50 percent (see Figure 5.4 for an example).

The remainder of this section gives a short introduction to the design considerations and syntax and semantics of the XABSL language. A complete language reference can be found on the XABSL website [69].

5.2.1 Symbol Definitions

Design considerations As described in Section 4.3 the latest version of XABSL supports several new symbol types. The set of symbol types is now more consistent than it was in previous version. There are three classes of symbols: input, output, and internal symbols. There are also three symbol data types: continuous (or decimal), binary (or boolean), and enumerated. Symbol classes and data types can be combined in each possible combination, thus there are nine different types of symbols. This enhances the versatility of the XABSL language. Furthermore constants can be defined for specifying frequently used values in a single place. There are also enumeration definitions which define the domains of the enumerated symbols.

All of the symbols have to be defined explicitly. This is necessary in order for the compiler to perform type checking. It was decided to use static types as this allows programming errors to be detected at compile time instead of at runtime only, thus supporting the development process.

Implementation specifics Similar to declarations in C++, before symbols can be referenced in options they have to be declared in advance. The symbol definitions are contained in separate symbol definition files. They can be grouped together thematically. Symbol definitions can therefore be distributed over several files. The elements of symbol definition files are definitions of input symbols, output symbols, internal symbols, constants, and enumerations.

Syntax definition To give an example, a symbol definition file, for instance called "my_symbols.xabsl", defining several symbols, enumerations and constants could look like this:

```
<state name="go_to_ball">
  <set-enumerated-output-symbol ref="head.control_mode">
    <enum-element-ref ref="head.control_mode.search_for_ball"/>
  </set-enumerated-output-symbol>
  <set-boolean-output-symbol ref="strategy.ball_is_grabbed">
    <boolean-value value="false"/>
  </set-boolean-output-symbol>
  <subsequent-option ref="go_to_ball"/>
  <decision-tree>
    <if>
      <condition description="ball can be grabbed">
        <and>
          <boolean-input-symbol-ref ref="ball.was_seen"/>
          <less-than-or-equal-to>
            <decimal-input-symbol-ref ref="ball.distance"/>
            <decimal-value value="142"/>
          </less-than-or-equal-to>
          <less-than>
            <decimal-input-symbol-ref ref="abs">
              <with-parameter ref="abs.value">
                <decimal-input-symbol-ref ref="ball.angle"/>
              </with-parameter>
            </decimal-input-symbol-ref>
            <decimal-value value="10"/>
          </less-than>
        </and>
      </condition>
      <transition-to-state ref="head_down"/>
    </if>
    <else>
      <transition-to-state ref="go_to_ball"/>
    </else>
  </decision-tree>
</state>
```

(a) A code fragment in the XML representation.

```
state go_to_ball
{
  decision
  {
    /** ball can be grabbed */
    if(ball.was_seen &&
            ball.distance <= 142 &&
            abs(value = ball.angle) < 10)
      goto head_down;
    else
      stay;
  }
  action
  {
    head.control_mode = search_for_ball;
    strategy.ball_is_grabbed = false;
    go_to_ball();
  }
}
```

(b) The same code fragment in the XABSL programming language.

Figure 5.4: Comparison of XML and XABSL code.

```
/** My most used symbols */
namespace my_symbols("My Symbols") {

  /** A boolean symbol */
  bool input something_wrong;

  /** A decimal symbol */
  float input foo "mm";

  /** The absolute value of a number */
  float input abs (
    /** The value for which the absolute value
        shall be calculated */
    float value;
  );

  /** Pets */
  enumeration type_of_pet {
    dog,
    cat,
    guinea_pig
  };

  /** Which pet was seen by the robot */
  enum type_of_pet input type_of_recognized_pet;

  /** Operation modes */
  enumeration op_mode {
   slow,
   fast
   very_fast
  };

  /** The mode how fast the robot shall act */
  enum op_mode output op_mode;

  /** The value of pi */
  float const pi = 3.14 "rad";

  ...
}
```

The general syntax of a symbol definitions file is the following:

```
<symbol definition file> ::=
  { include "<include file>"; }
```

```
namespace <id>("<title>")
{
  {
    <enumeration> | <input symbol> |
    <output symbol> | <internal symbol> | <constant>
  }
}
```

The field *id* contains the name of the symbol collection which must be identical to the file name, while *title* contains a longer description of the symbol collection. Other symbol definition files can be included as they can contain enumeration definitions referenced by some of the symbol definitions.

The following sections will describe the five possible types of symbol definitions: enumerations, inputs symbols, output symbols, internal symbols, and constants.

5.2.1.1 Enumerations

Design considerations The goal is to define an enumeration of elements used in enumerated symbols. User defined enumerations can be used as data types for symbols and parameters. The list of enumeration elements describe the possible discrete values an enumerated symbol of this type can assume. Enumeration element names do not necessarily have to be unique. When enumeration elements appear in expressions it is always guaranteed that the enumeration they are associated with is unambiguous. How this is realized is described below in Section 5.2.10. This is motivated by the fact that using the same enumeration element name in different enumerations may very well be justified. (An example from robot soccer: An enumeration *goalColor* might have the elements *blue* and *yellow*, while another enumeration *teamColor* may have the elements *blue* and *red*.) If the enumeration element names had to be made unique, for instance by prepending the enumeration name, this would only serve to clutter the source code unnecessarily.

Syntax definition

```
<enumeration> ::=
  enum <name>
  {
    <enum-element>
    { , <enum-element> }
  };
```

5.2.1.2 Input symbol

Design considerations An input symbol can have one of three types: real-valued (called decimal) specified by keyword *float* (corresponds per default to double precision floating point values in the C++ implementation of the XABSL engine but single precision values could be used as well), boolean, or enumerated. *Range* and *measure* are additional data specified for automatically generated documentation. In case the type of the symbol is enumerated, the name of the corresponding enumeration has to be specified and the enumeration has to be defined beforehand.

All input symbols can be parameterized. While an input symbol without parameters gives access to a single variable, parameterized input symbols allow access to arbitrary functions in the software environment. For instance, parameterized input symbols can provide complex operations that are otherwise not feasible with simple expressions. In previous versions of XABSL parameterized symbols could only be realized through the specific *function* construct. Having the possibility to add parameters to any input symbol has increased the expressiveness as previously functions were not available for all types of input symbols. At the same time the language could be simplified by removing the no longer required *function*. Again the fields *range* and *measure* are only required for documentation. In case of an enumerated parameter the corresponding enumeration must be specified.

Syntax definition

```
<input symbol> ::=
  [float] [input] <name> [[<range>]] ["<measure>"] |
  bool [input] <name> |
  enum <enumeration> [input] <name>
  [
  (
     <parameter>
  )
  ]
  ;
```

An input symbol can have decimal, boolean, or enumerated parameters, and they are defined using the following syntax:

```
<parameter> ::=
  [float] <name> [[<range>]] ["<measure>"] |
  bool <name> |
  enum <enumeration> <name>
  ;
```

5.2.1.3 Output symbols

Design considerations Unlike in previous versions of XABSL the current value of an output symbol can also be queried in expressions. This means that output symbols can be used in the same way as unparameterized input symbols. This feature was added so that it is no longer required to map the value of an output symbol to an input symbol, when it was necessary to query the current value of an output symbol.

Syntax definition

```
<output symbol> ::=
  [float] output <name> [[<range>]] ["<measure>"] |
  bool output <name> |
  enum <enumeration> output <name>
  ;
```

5.2.1.4 Internal symbols

Design considerations Internal symbols can be treated in the same way as output symbols. The only difference is that internal symbols do not need to have an representation outside of the XABSL state machine. Internal Symbols were added to XABSL recently. They do not offer new functionality since it was always possible to emulate the behavior by mapping the value of an output symbol to an input symbol. Since the feature has been added that the current values of output symbols can be queried in expressions it is no longer necessary to map output to input symbols. With the addition of internal symbols not even the registration and mapping to a variable outside of the XABSL engine is required anymore. Thus, it has become very easy to implement symbols to be used for data exchange between options or for storing continuous state variables.

Syntax definition

```
<internal symbol> ::=
  [float] internal <name> [[<range>]] ["<measure>"] |
  bool internal <name> |
  enum <enumeration> internal <name>
  ;
```

5.2.1.5 Constants

Design considerations Constants are useful for defining frequently used values in only one place. Using constants can help to easily adapt certain parameter values and prevents errors resulting from inconsistent values. The use of constants is not required as they do not add functionality which would otherwise not be accessible, but they support the developers in producing well maintainable source code.

Syntax definition

```
<constant> ::=
  [float] const <name> = <value> ["<measure>"]
  ;
```

5.2.2 Basic Behavior Definitions

Implementation specifics For each basic behavior, a prototype has to be declared. Basic behavior prototypes are contained in a basic behavior definition file. There can be more than one basic behavior definition files in order to group basic behaviors.

Syntax definition A basic behavior definition file (*"my_basic_behaviors.xabsl"*) could, for example, look like this:

```
/** My common basic behaviors */
namespace my_basic_behaviors("My Basic Behaviors") {
  /** Lets the agent move to a point */
  behavior move_to {
    /** X of destination position */
    float x [-1000..1000] "mm";
    /** Y of destination position */
    float x [-1000..1000] "mm";
  };
}
```

The general syntax of a basic behavior definition file is defined as follows:

```
<basic behavior definition file> ::=
  { include "<include file>"; }
  namespace <id>("<title>")

    { <behavior> }
```

The field *id* contains the name of the basic behavior collection which must be identical to the file name, while *title* contains a longer description of the basic behaviors. Other symbol definition files might need to be included as they can contain enumeration definitions referenced by basic behavior parameters.

The syntax of the definition of basic behavior prototypes is as follows:

```
<behavior> ::=
  behavior <name>
  [

    { <parameter> }

  ]
  ;
```

Optionally a number of parameters can be defined. The definition of parameters of basic behavior is identical to the definition of input symbol parameters. Decimal, boolean, or enumerated parameters are defined using the following syntax:

```
<parameter> ::=
  [float] <name> [[<range>]] ["<measure>"] |
  bool <name> |
  enum <enumeration> <name>
  ;
```

5.2.3 Options

Syntax definition Each option has to be defined in a separate file, e.g. "Options/foo.xabsl":

```
include "../my_symbols.xabsl"
include "../my_basic_behaviors.xabsl"

/** Some option */
option foo {
  bool @aParameter;

  initial state first_state {
    ...
  }

  state second_state {
    ...
  }
}
```

An option file must have the following syntax:

```
<option file> ::=
  { include "<include file>"; }
  option <name>
  {
    { <parameter> }
    [ <common decision tree> ]
    <state>
    { <state> }
  }
```

The *name* of the option must be identical to the file name. Other files have to be included which contain definitions of any symbols used in the option, and of other options and basic behaviors that are referenced in the actions of the option. Options can have parameters which are defined in the same way as input symbol parameters. The name of an option parameter has to start with "@" in order to be distinguished from an input symbol:

```
<parameter> ::=
  [float] @<name> [[<range>]] ["<measure>"] |
  bool @<name> |
  enum <enumeration> @<name>
  ;
```

5.2.4 Common Decision Trees

Design considerations If there are transitions with the same conditions in each state of an option, these conditions can be put into the common decision tree. It is carried out before the decision tree of the active state. If no condition of the common decision tree evaluates to true, the decision tree of the active state is carried out. That is also the reason why there must be no else statement which is not followed by another if statement in the common decision tree.

Common decision trees can be used in order to eliminate redundancy in the behavior implementation when the states of an option have similar or identical transition conditions. Therefore, analogously to the definition of constants (cf. Section 5.2.1), while this feature does not increase the expressiveness of the language, it helps the developers to produce readable and maintainable source code. Often it is even the case that the decision trees of all of the states of an option are identical. In that case the common decision tree can be used to describe the complete transition graph. The individual decision trees of the states can be omitted. The decision making of the option then is no longer state based. The state machine is degenerated as the next state no longer depends on the current state but only on the current inputs queried in the common decision tree. This can be useful at certain levels of the behavior hierarchy where the decision making is purely rule based. This special case is comparable with similar concepts in other behavior architectures, for instance, competitive assemblages in the *Configuration Description Language* (cf. Section 2.5).

Implementation specifics If the common decision tree contains expressions that are specific for a state (*state_time*, *action_done*), these expressions refer to the currently active state.

The elements boolean expression and decision tree are the same as in the normal decision tree of a state, which is explained later in this document.

For details of the syntax and semantics of the common decision tree, see the description of decision trees in Section 5.2.6.

Syntax definition A common decision tree can be defined optionally at the start of an option with the following syntax:

```
<common decision tree> ::=
  common decision
  {
    if ( <boolean expression> )
      <decision tree>
    {
      else   if ( <boolean expression> )
        <decision tree> |
      else { if ( <boolean expression> ) }
        <decision tree>
    }
  }
```

After the optional common decision tree each option has to have at least one state definition, which is described in the next section.

5.2.5 States

Design considerations If an option executes another option, it can be queried from the calling option whether the subsequent option reached a marked target state. Querying whether a target state has been reached is the only possibility for a calling option to gain feedback from the callee, comparable to a binary return value. It was decided that there is no more elaborate method of giving feedback to the calling option, as this very simple mechanism, which, for instance, allows an option to report whether its given task has been carried out completely, is sufficient in most applications. If more than binary feedback is required, this can also be easily realized using an internal symbol.

Syntax definition A definition of a single state of an option's state machine is shown in this example:

```
initial target state first_state {
  decision {
    if (foo < 14 )
      goto second_state;
    else
      goto first_state;
  }
  action {
    move_to(x = 42);
    op_mode = fast;
  }
}
```

A state definition has the following syntax:

```
<state> ::=
  [initial] [target] state <name> {
    decision
    {
      [ else ] <decision tree>
    }
    action
    {
      { <action definition> }
    }
  }
```

If the keyword *initial* is specified, the state is marked as the initial state of the option. This must be set for exactly one state in the option.

If the keyword *target* is specified, this state is marked as a target state.

The *decision tree* defines the transitions of the state. Its syntax described in the next section. The leading else before the decision tree must be specified if and only if the option has a common decision tree. This reflects the fact, that when the option has a common decision tree, the decision tree of a state is only executed when no transition was selected by the common decision tree.

An *action definition* defines an action to be executed when the state is active. Each state can have any number of action definitions.

5.2.6 Decision Trees

Design considerations Each state can have a decision tree. The decision tree describes how to determine a transition to another state depending on the input symbols.

The syntax of decision trees was selected to be analogous to that of *C if/else*-statements. This makes them intuitively understandable by most developers. The main difference however is that *else* statements cannot be omitted. The purpose of a decision tree is to select exactly one of the states to be the active state in the next execution cycle.

Syntax definition The syntax of a decision tree is as follows:

```
<decision tree> ::=
    { <decision tree> }
  |
    if ( <boolean expression> )
    <decision tree>
    else
    <decision tree>
  |
    goto <state>;
  |
    stay;
```

Implementation specifics A decision tree contains either an *if/else* block or a transition to a state.

The *if/else* element consists of a boolean expression and two decision trees. The first one is executed if the expression evaluates to true, the second one otherwise. This recursive definition allows for complex nested conditions. A transition to a specified state is given by a *goto* statement. The statement *stay* represents a transition to the current state. When this transition is executed, the active state of the option remains unchanged in the next execution cycle.

If the decision tree is omitted, an empty decision tree is assumed which always selects the current state.

5.2.7 Action Definitions

Design considerations In previous XABSL versions, each state could only activate exactly one option or basic behavior and optionally set a number of enumerated output symbols. As described in Section 4 in order to meet the design goals this has been enhanced. Since concurrent execution is required it is now possible to specify any number of actions which are executed concurrently. Furthermore, since another requirement is to allow continuous outputs, outputs are no longer restricted to enumerated symbols. There can now also be continuous outputs in the form of decimal output symbols and boolean outputs for the sake of completeness.

Syntax definition Each state has a number of action definitions. These definitions specify which actions are to be executed when a state is active. The syntax is as follows:

```
<action definition> ::=
    <output symbol> | <internal symbol>
    =
    <decimal expression> |
    <boolean expression> |
    <enumerated expression>
    ;
  |
    <option>  [ <parameter list> ] ;
  |
    <basic behavior> [ <parameter list> ] ;
```

When options or basic behaviors are referenced, a list of parameters can be specified. This is done according to the following syntax:

```
<parameter list> ::=
  (
    <parameter> =
    <decimal expression> |
    <boolean expression> |
    <enumerated expression>
    {
      , <parameter> =
      <decimal expression> |
      <boolean expression> |
      <enumerated expression>
    }
  )
```

Implementation specifics An action definition might contain an assignment to an output or internal symbol. When the state is active, the value of the symbol is set to the value of the given expression. The expression must be of the same type as the symbol. This is checked at compile time. It can happen that the symbol value gets overwritten from another option even in the same execution cycle.

An action definition can also be a reference to another option or a basic behavior which is to be executed when this state is active.

The value of each parameter is set to the value of the given expression. The expression must be of the same type as the parameter. Mismatching types will cause a compilation error. If not all parameters of an option or basic behavior are set, the executing engine sets the remaining parameter values to zero.

Note that in each execution cycle the decision tree is evaluated first and only the actions of the resulting next current state are then executed afterwards.

5.2.8 Boolean Expressions

Design considerations Boolean expressions are required in decision trees. Furthermore boolean expressions are used to parameterize symbols, options, and basic behaviors, and in order to be assigned to boolean output symbols.

Syntax definition A boolean expression can have the following syntax:

```
<boolean expression> ::=
    ( <boolean expression> ) |
    !<boolean expression> |
    <boolean expression> && <boolean expression> |
    <boolean expression> || <boolean expression> |
    <qualified enumerated expression> ==
      <enumerated expression> |
    <qualified enumerated expression> !=
      <enumerated expression> |
    <decimal expression> == <decimal expression> |
    <decimal expression> != <decimal expression> |
    <decimal expression> < <decimal expression> |
    <decimal expression> <= <decimal expression> |
    <decimal expression> > <decimal expression> |
    <decimal expression> >= <decimal expression> |
    <boolean input symbol> [ <parameter list> ] |
    <boolean output symbol> |
    <boolean internal symbol> |
    @<boolean option parameter> |
    true |
    false |
    action_done
```

Boolean expressions have the following semantics:

- "!": Boolean not operator. Inverts a boolean expression.
- "&&" and "——": Boolean and/or operator. Combines two boolean expressions.

- "==" and "!=": Compares two decimal or enumerated expressions. Please note that in case of enumerated the expression the left-hand side expression must be qualified, i.e. it must specify the enumeration. Enumerated expressions are described below in this document.
- "¡", "¿", "¡=", "¿=": Compares two decimal expressions.
- "*boolean input symbol*": References a boolean input symbol. Optionally parameters can be specified. See previous section for syntax of parameter list.
- "*boolean output symbol*": References a boolean output symbol. This queries the value set to the output symbol previously.
- "*boolean internal symbol*": References a boolean internal symbol. This queries the value set to the internal symbol previously.
- "*boolean option parameter*": References a boolean option parameter.
- "action_done": This expression becomes true when the current state has a subsequent option and the active state of the subsequent option is marked as a target state. Otherwise this statement is false. This can be used to query feedback from subsequently executed options, whether their execution has been finished or not.

5.2.9 Decimal Expressions

Design considerations Decimal expressions can be used inside some boolean expressions for parameterizing symbols, options, and basic behaviors, and for the assignment of decimal output symbols.

Syntax definition They have the following syntax:

```
<decimal expression> ::=
    ( <decimal expression> ) |
    <decimal expression> + <decimal expression> |
    <decimal expression> - <decimal expression> |
    <decimal expression> * <decimal expression> |
    <decimal expression> / <decimal expression> |
    <decimal expression> % <decimal expression> |
    <boolean expression> ? <decimal expression> :
                           <decimal expression> |
    <decimal input symbol> [ <parameter list> ] |
    <decimal output symbol> |
    <decimal internal symbol> |
    @<decimal option parameter> |
    <constant> |
    <decimal value> |
    state_time |
    option_time
```

Decimal expressions have the following semantics:

- "+","-","*","/" and "%": Arithmetic +, -, *, / and % operators.
- "*boolean expression*"?"*decimal expression*":"*decimal expression*": Defines a conditional expression, which works like an ANSI C question mark operator. If the boolean expression is true, the left-hand side decimal expression is returned. Otherwise the right-hand side decimal expression is returned.
- "*decimal input symbol*": References a decimal input symbol. Optionally parameters can be specified. See previous section for details on syntax of parameter lists.
- "*decimal output symbol*": References a decimal output symbol. This queries the value set to the output symbol previously.
- "*decimal internal symbol*": References a decimal internal symbol. This queries the value set to the internal symbol previously.
- "*decimal option parameter*": References a decimal option parameter.
- "*constant*": References a constant defined in a symbol definition file.
- "*decimal value*": A decimal value, e.g. "3.14".
- "state_time": This expression returns the duration in seconds for which the current state of the option is active. Whenever a state change occurs this time is set to zero. If the activation tree that lead to activating the option changes, but the option remains active, the time is not reset. It only matters whether the option and the state were active in the previous execution cycle.
- "option_time": This expression returns the duration in seconds for which the current option is active. The time will also not reset when the activation tree has changed, as long as the option remains active.

5.2.10 Enumerated Expressions

Design considerations Enumerated expressions can be used inside some boolean expressions for parameterizing symbols, options, and basic behaviors, and for the assignment of enumerated output symbols. For the left-hand side of the comparison of two enumerated expressions in a boolean expression a so-called qualified enumerated expression is required which implicitly defines its associated enumeration. Particularly a reference to an enumeration element cannot be used, since enumeration elements are not necessarily unique. E.g. the boolean expression "*dog == type_of_recognized_pet*" would be illegal; "*type_of_recognized_pet == dog*" must be used instead.

Syntax definition Enumerated expressions follow this syntax:

```
<qualified enumerated expression> ::=
    ( <qualified enumerated expression> ) |
    <enumerated input symbol> [ <parameter list> ] |
    <enumerated output symbol> |
    <enumerated internal symbol> |
    @<enumerated option parameter> |
    <boolean expression> ?
      <qualified enumerated expression> :
      <qualified enumerated expression>

<enumerated expression> ::=
    <qualified enumerated expression> |
    <boolean expression> ? <enumerated expression> :
                           <enumerated expression> |
    <enumeration element>
```

Enumerated expressions have the following semantics:

- "*enumerated input symbol*": References an enumerated input symbol. Optionally parameters can be specified.
- "*enumerated output symbol*": References an enumerated output symbol. The value previously set to the output symbol is queried.
- "*enumerated internal symbol*": References an enumerated internal symbol. The value previously set to the internal symbol is queried.
- "*enumerated option parameter*": References an enumerated option parameter.
- "*boolean expression*""?""*enumerated expression*"":""*enumerated expression*": Defines a conditional expression, which works such as an ANSI C question mark operator. If the boolean expression is true, the left-hand side enumerated expression, otherwise the right-hand side enumerated expression, is returned.
- "*enumeration element*": References an enumeration element.

5.2.11 Agents

Design considerations The agents definition file ("*agents.xabsl*") is the root document of an XABSL behavior specification. It includes all the necessary options and defines agents.

In an XABSL behavior specification, the option graph does not need to be completely connected. So it is not possible to determine a single root option of the graph. Instead a sub-graph that is spanned by an option and all it's subsequent options and basic behaviors can be declared as an agent. Therefore, an agent defines a starting point into the option graph.

Syntax definition The agents definition file has the following syntax:

```
<agents definition file> ::=
  { include "<include file>"; }
  { agent <id>("<agent-title>", <root-option>); }
```

An example "agents.xabsl" file may look like this:

```
/***
  Title: My XABSL behavior application
  Platform: My robot/agent platform.
  Software-Environment: My software platform
*/

include "Options/foo.xabsl";
include "Options/bla.xabsl";

/** The default agent */
agent default_agent("Default", foo);

/** A test environment for the option bla */
agent test_behavior("Test", bla);
```

Implementation specifics Included files must contain option definitions for options to be used as agent root options.

There has to be at least one agent definition element inside the agents definition file. It contains an identifier of the agent, a title which is required for documentation only, and the name of the root option of the agent.

5.3 Compiler

In the context of this thesis the existing XABSL compiler written in Ruby [92] has been enhanced in order to support all of the new features of the improved version of the XABSL architecture.

The compiler can generate different types of documents from an XABSL document: intermediate code which is interpretable by the runtime system, a list of keywords for code completion and syntax highlighting for a variety of editors, and an XML representation of XABSL specifications.

The XML representation can easily be parsed by supporting tools e.g. an XSLT processor can be used to generate an extensive HTML documentation containing SVG (Scalable Vector Graphics) charts for the option graph, each option, and each state.

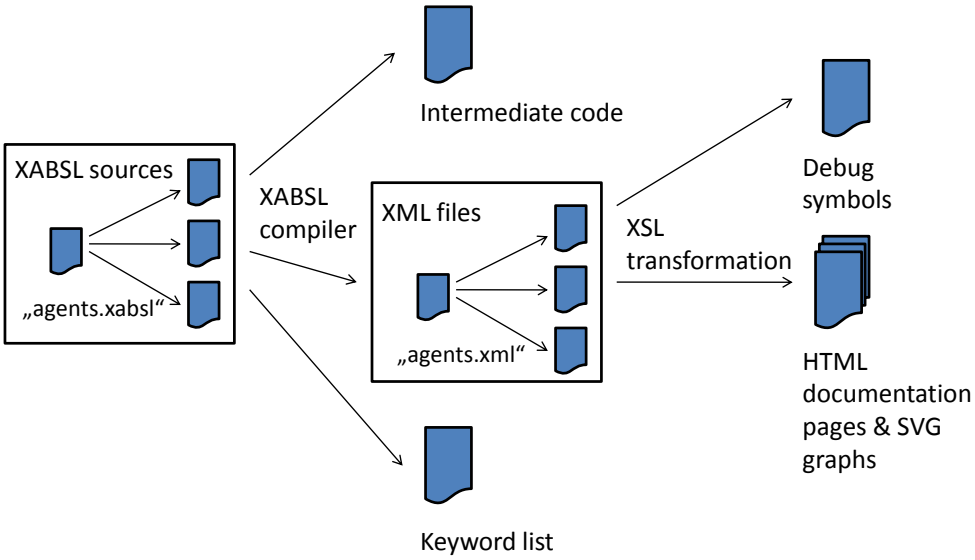


Figure 5.5: An overview of document processing in XABSL.

There is also an XSL transformation which generates a file with symbolic debug information containing names of symbols, options, basic behaviors, and parameters. This debug symbol file can be used in order to create tools that connect to an agent running the XABSL engine for instance for monitoring. However, this is not the recommended method of gaining debug information, as the XABSL engine also provides means of obtaining this information directly from an agent at runtime. The latter method eliminates the need of recompiling anything besides the intermediate code when source code has been changed. If the agent provides a method of reloading the intermediate code at runtime, very quick debugging cycles can be realized. Reading the debug information directly from the agent also prevents version conflicts, for instance, when monitoring an agent running an older software version. If the symbol names are read from a file, it would be required to know in advance which agent is running in which version in order to provide the correct symbol files.

Figure 5.5 gives an overview of the different types of documents that can be generated out of the XABSL source code, and how they are generated.

Note that all of the figures in this thesis that contain XABSL options or option graphs were generated automatically from XABSL sources.

5.4 Runtime System

The class library *XabslEngine* is the XABSL runtime system. There are versions of the engine in plain ANSI C++ and newly also in Java and they are platform and application independent. To run the engine in a specific software environment, only mechanisms for file access and error handling have to be adapted to the target platform. The engine parses and executes the intermediate code that was generated from XABSL documents using the XABSL compiler. It links the symbols from the XABSL specification that are used in the options and states to the variables and functions of the agent platform. Therefore,

for each used symbol an entity in the software environment is registered to the engine. Basic behaviors are written in C++ or Java and also registered to the engine at startup. The class library provides extensive debugging interfaces for monitoring and manipulating nearly all internal states of the engine. A complete API documentation is available at the XABSL web site [69].

Based on the engine's debugging interfaces it is easy to develop tools which can display the option activation path, the parameters and execution times of options, states, and basic behaviors, as well as the values of input and output symbols. Using the debugging interface it is also possible to manually select and parameterize single agents, options, or basic behaviors for execution.

The tools for debugging and logging described in the next section were implemented using this debugging interface.

5.5 Tools for Development

Some tools which facilitate working with XABSL are available from the website [69]. Other tools are part of the software developed in the *GermanTeam* and are also partially available online [86].

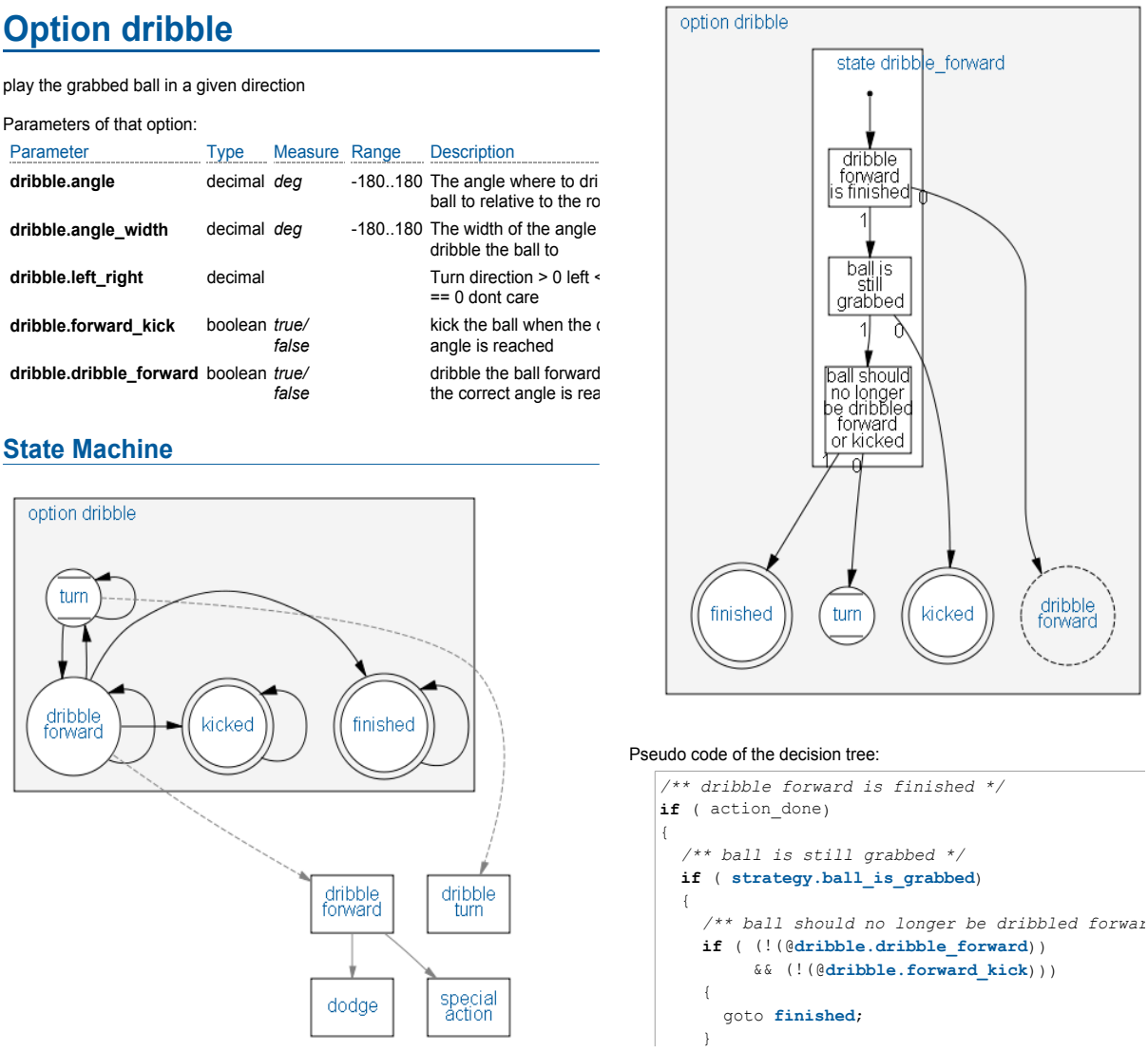
5.5.1 Documentation

The XABSL release includes XSL templates which can be used to generate HTML documentation containing SVG diagrams by parsing the XML representation generated from XABSL sources. Similar to, for instance, *JavaDoc* the XABSL compiler interprets certain comments in the source code as descriptions that will be included in the XML representations and generated documentation pages. As the source code, the generated documentation is distributed over many files. Therefore, in order to speed up the compilation of the documentation, it is necessary to rebuild only those documentation pages affected by changes in the source code. In order to be able to do so, there are different XSL templates for the different documentation pages and a makefile is used to invoke the documentation generation after the source code has been changed. The documentation contains SVG graphics with visualizations of options graphs, state machines, and decision trees. These graphics are generated using *dot* from the *GraphViz* software suite [26, 36]. The dot sources are generated using *DotML* an XML wrapper for automatically generating the dot input format [66].

Figure 5.6 shows an example of an automatically generated option documentation page including a graph of the state machine and an example of a decision tree of one of the states.

5.5.2 XABSL Editor

There is an editor for the XABSL programming language, which has been developed by members of the *GermanTeam* [87]. It provides a graphical display of XABSL options, showing state diagrams and option graphs. It allows behaviors to be browsed in the editor window. By clicking on options or states in the graphical display, the respective source



Option dribble

play the grabbed ball in a given direction

Parameters of that option:

Parameter	Type	Measure	Range	Description
dribble.angle	decimal	*deg*	-180..180	The angle where to dri ball to relative to the ro
dribble.angle_width	decimal	*deg*	-180..180	The width of the angle dribble the ball to
dribble.left_right	decimal			Turn direction > 0 left < == 0 dont care
dribble.forward_kick	boolean	*true/ false*		kick the ball when the (angle is reached
dribble.dribble_forward	boolean	*true/ false*		dribble the ball forward the correct angle is rea

State Machine

Pseudo code of the decision tree:

```
/** dribble forward is finished */
if ( action_done)
{
  /** ball is still grabbed */
  if ( strategy.ball_is_grabbed)
  {
    /** ball should no longer be dribbled forwar
    if ( (!(@dribble.dribble_forward))
         && (!(@dribble.forward_kick)))
    {
      goto finished;
    }
```

Figure 5.6: An HTML option documentation page.

code will be opened. The editor also provides syntax highlighting and source completion. In order to update the display, the XABSL compiler is invoked directly from the editor. The compiler generates XML representations which are used to generate the displayed graphics. Errors reported from the compiler will also be displayed directly in the editor.

When XABSL code was written in XML, standard tools for XML editing could be used. Since there is the XABSL programming language, this option is no longer available. Therefore, an editor was required to support the user in quickly editing behavior source code. The XABSL editor provides a convenient way of doing so. Especially inexperienced users benefit from having a graphical display which immediately visualizes changes to the state machine.

See Figure 5.7 for a screenshot.

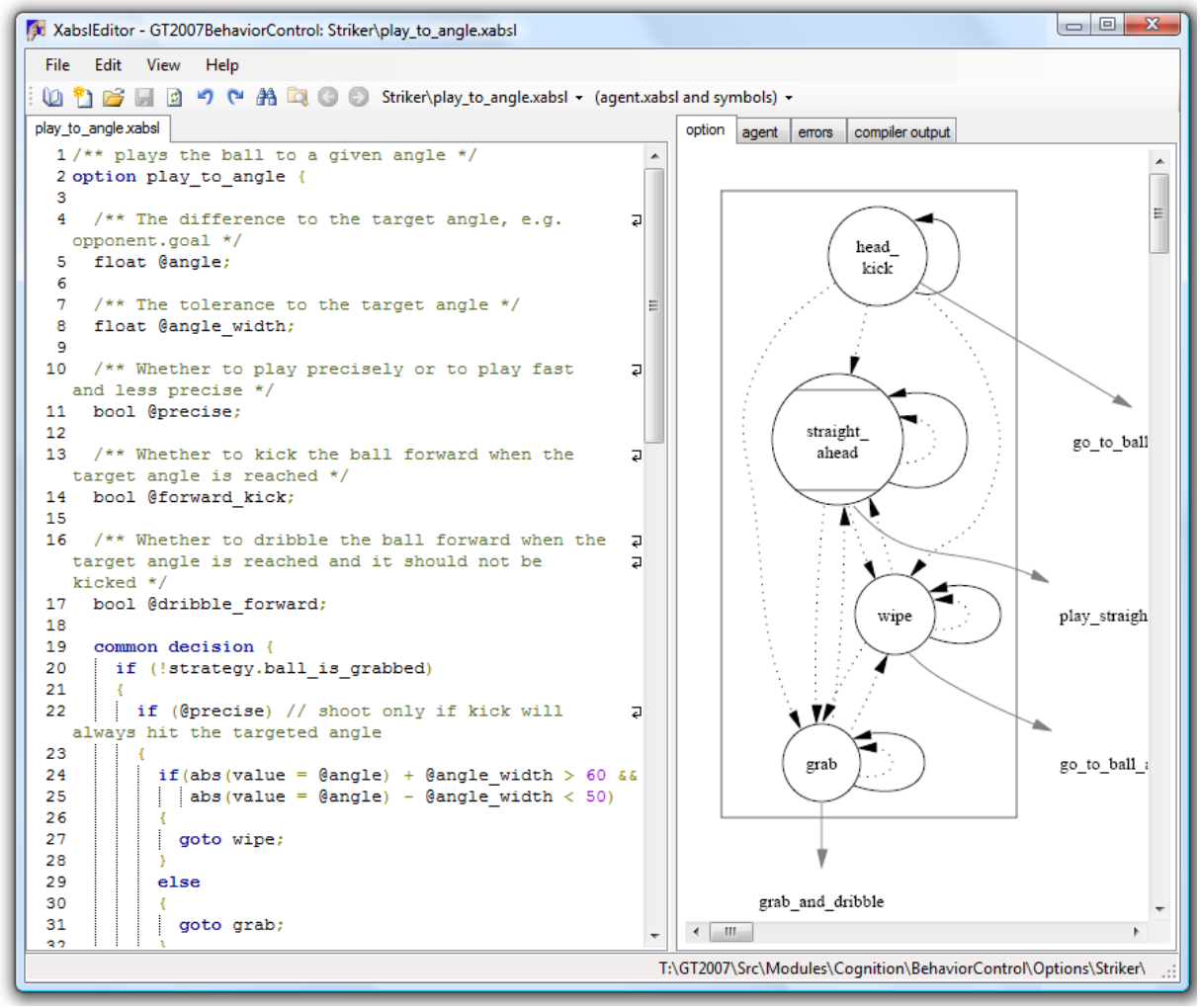


Figure 5.7: A screenshot of the XABSL editor. The left hand window shows the source code, while the right hand window shows a visualization of the current option, including their state, transitions, and actions.

5.5.3 Monitoring

By using the debugging interfaces of the XABSL engine, application specific debugging and monitoring tools can easily be obtained. Fig. 5.8 shows an example of the XABSL dialog integrated in the debugging tool used by the *GermanTeam* for *RoboCup* 2007.

When dealing with complex agents it is important to have such monitoring tools in order to keep track of the internal states and variables. When the agent exhibits unwanted behavior, in general the cause cannot be determined with reasonable certainty merely by observing the agent. Therefore, in order to find behavior errors or inadequacies, having insight into the behavior execution can prove to be an important tool. Monitoring the tree of active options during a situation in which the agent behaves incorrectly might help to narrow down the responsible behavior options. Monitoring input symbol values might reveal whether the cause of a problem lies in the behavior programming or rather in the inaccuracy of input data. For instance, in a robot soccer scenario, when a robot is

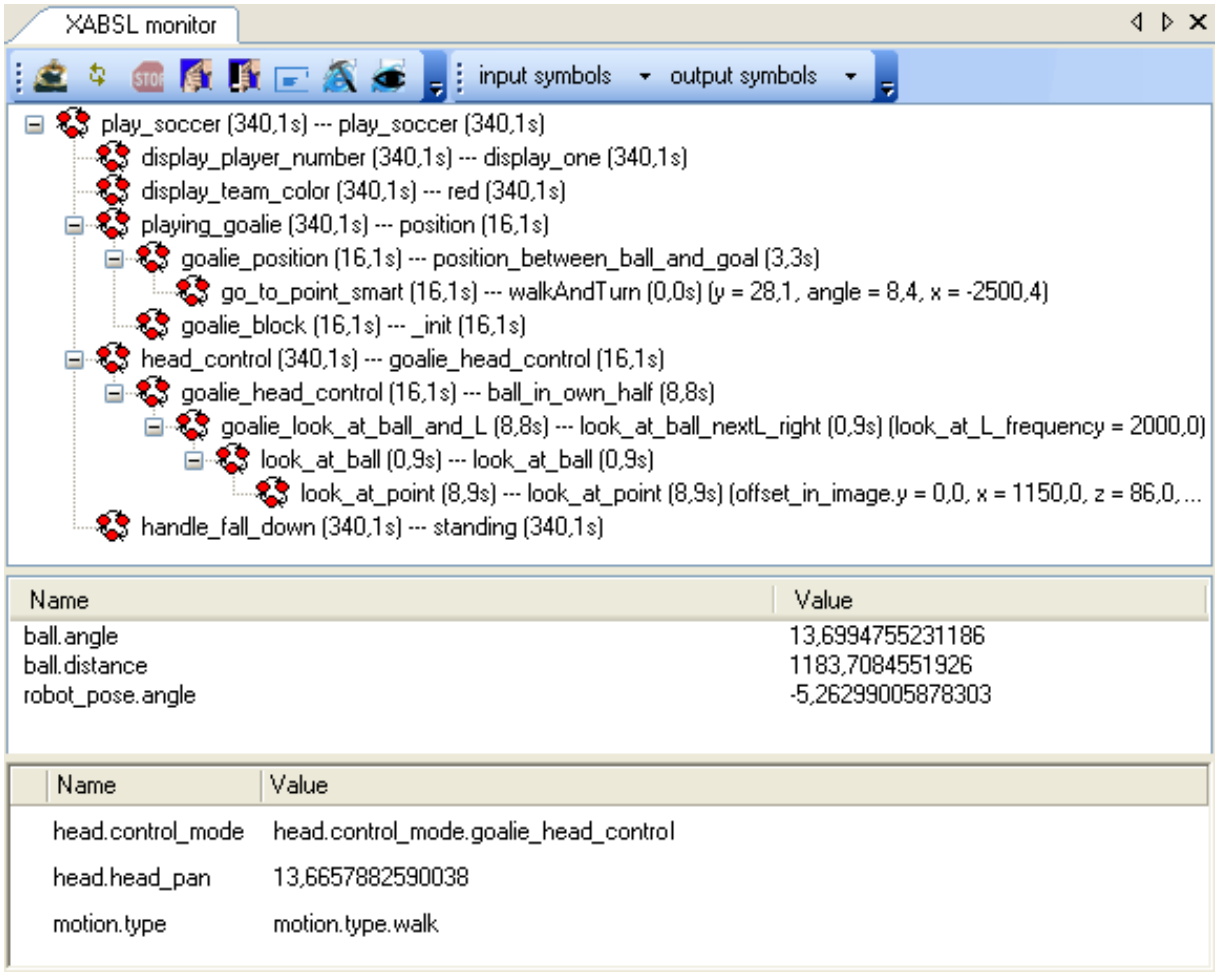


Figure 5.8: An example for a runtime monitoring tool realized by using the debugging interfaces of the XABSL engine. The upper section of the dialog shows the current state of the engine, i.e. the option activation tree, the respective current states of the active options, and current option parameter values. The middle and bottom sections monitor the current values of selected input and output symbols.

playing the ball in the direction of the own goal, this can either be caused by incorrect behavior programming or by drastically incorrect self-localization. The cause cannot be determined simply by watching the robots play.

5.5.4 Logging

Even more important than the ability to monitor agent behavior during runtime is the ability to analyze behavior in retrospect. Ideally log files are recorded at all times during the operation of an agent, so that any erroneous behavior can be analyzed afterwards. Of course, the same debugging interface provided by the XABSL engine for querying activations and symbol values, which is used to realize monitoring features, can also be used for implementing logging. In real robot applications logging can be severely limited by different practical implications. For instance, memory required for log data can be an issue.

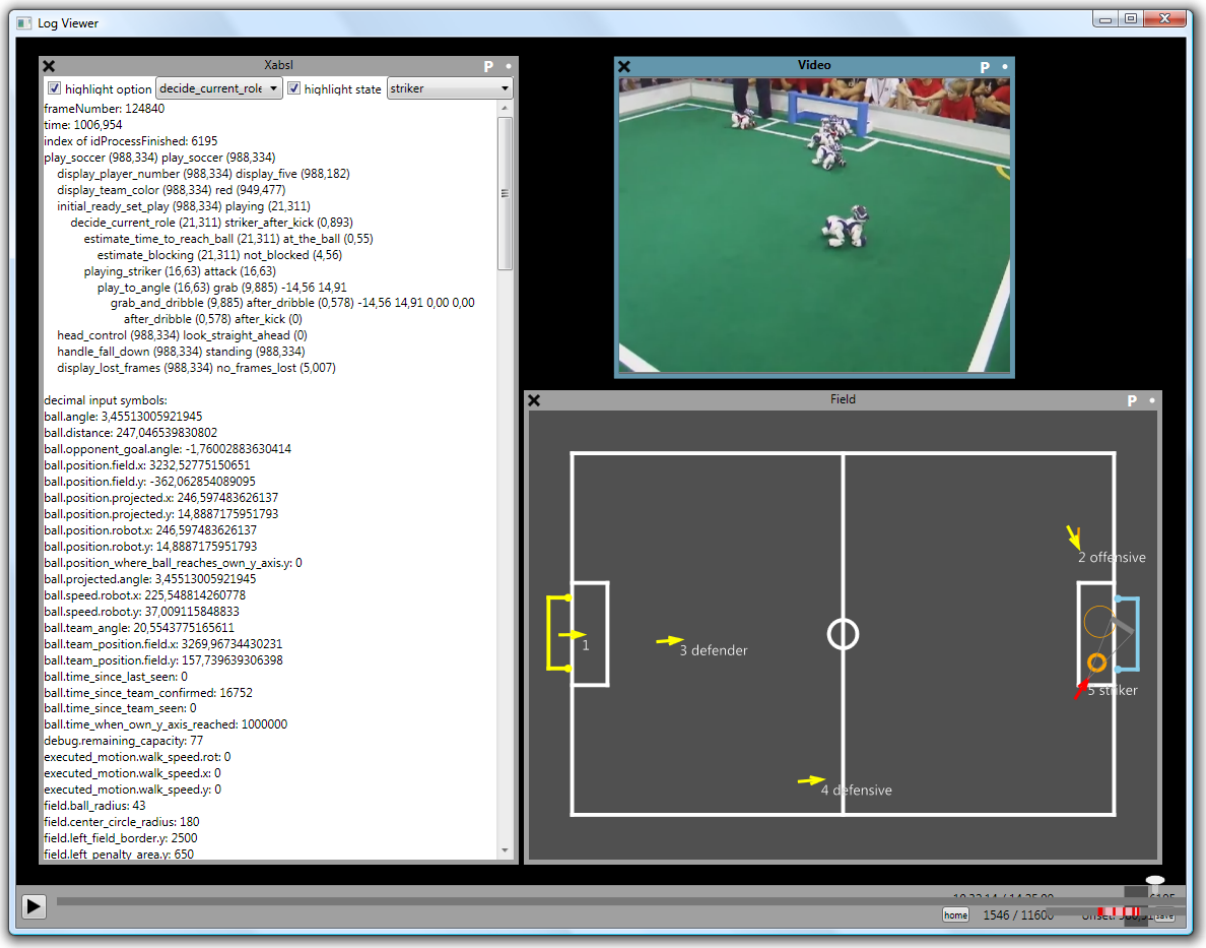


Figure 5.9: The log viewer developed by the *GermanTeam*. One window shows the current behavior data, e.g. the activation tree and input symbol values. Another gives a visualization of some of the input symbols, showing on the soccer field the position of the robot, the position of the ball, the teammates and their roles. In a third window a video can be replayed synchronously with the recorded log data.

Often log data has to be stored in working memory or in limited persistent storage space. Under no circumstances should the recording of log data have a negative effect on the performance of the system. As computational power is usually very limited in autonomous systems, expensive compression methods are not available. A compromise must therefore be found balancing the information content required for allowing meaningful analysis and the amount of data which can be logged.

In the robot soccer application developed by the *GermanTeam* powerful logging facilities helped the team to identify problems not only limited to behavior control. The input symbols provided to the behavior already provide a compact view on the relevant data generated by world modeling modules. Therefore, logging all input symbol values already provides enough information in order to evaluate the performance of world modeling. Especially when comparing recorded log data with video footage, incorrect information and inaccuracies in the input data can be identified.

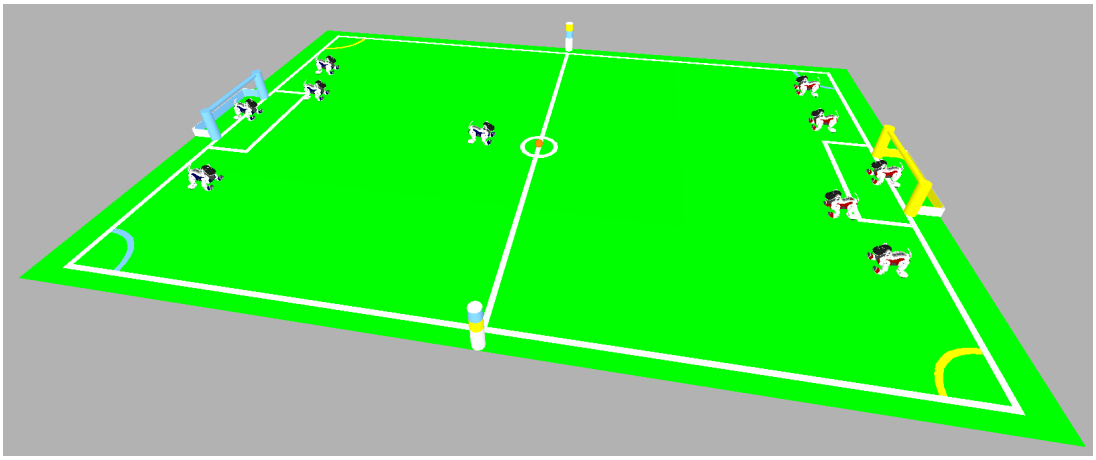


Figure 5.10: The simulator used by the *GermanTeam*.

The log files recorded at *RoboCup* 2008 in every game contain all behavior symbol values, active options, states, and parameters. In order to reduce log size only the difference from the previous frame is stored. As only a small number of input symbols change their values in each frame, this leads to a large reduction in log data size. Changes in option activation are also stored differentially in the same way. Since the active options and their states do not change in every cycle, this also provides good compression of log data at minimal expense.

A viewer for log files has been developed which is used to visualize the recorded data. The viewer provides visualizations for some of the symbol values, such as robot positions. Also the option activation tree for each frame can be displayed. Certain option or state activation can be displayed on a time bar, thus enabling the developer to quickly find specific situations in the log data. Comparison with video footage is also supported: A video can be loaded and played back synchronously with the log data. When video footage is available, synchronizing video and log data can be troublesome. This is currently done manually by identifying certain situations in the log data, such as kick-offs or goals, which can also be found in the video. Once the time offset between log file and video has been determined it can be stored for each log file.

Figure 5.9 shows the log viewing tool.

5.5.5 Simulator

An essential tool in programming any real robot application is a simulator which allows the developers to perform tests without having to rely only on real hardware. Being able to test in a simulation can help to reduce development time and cost tremendously, as conducting experiments on real hardware can be very time consuming and expensive.

Especially when programming behavior control, having a simulation tool available can prove very helpful. The requirements for testing behavior in a simulation are lower than for testing other algorithms since an exact simulation of sensor data is not necessarily required. For instance, when testing image processing a photo-realistic simulation of the camera system might be required for the results to be comparable with results obtained with real images. When solely testing behavior it is not even required that the actual

world modeling modules are in operation. Instead, it can also be considered obtaining word model data directly from the simulation, optionally including simulated noise or uncertainties. Testing behavior on perfect simulated data not containing any uncertainties can be a very interesting scenario, as it allows the programmers to test whether certain behaviors are principally working as intended. Doing this as a first test could be considered for any newly developed behavior. If an agent already behaves incorrectly when provided with undisturbed input data, it will never work correctly when noise and uncertainty are added. On the other hand, a successful test on perfect data does not eliminate the need to test with a more exact simulation including realistic noisy inputs or on real hardware, as the behavior might rely on unrealistic accuracy requirements.

Of course the availability of a simulation tool is not only desirable when developing behavior with XABSL. This applies to the same degree for any other behavior architecture.

The *GermanTeam* uses the simulator *SimRobot* [63] in order to simulate a robot soccer scenario. Figure 5.10 shows a screenshot of the simulator.

6 Applications and Results

Now that the improvements to the XABSL architecture and its implementation have been presented in the previous chapters, this chapter presents some of the applications that are realized using XABSL. It is evaluated by means of different applications examples whether the targeted design goals were met.

Since *RoboCup* is the primary test bed, most of the applications come from the robot soccer domain. But also some applications from different domains are presented.

6.1 Evaluation of Design Goals on the Basis of Different Applications

In this section it is evaluated whether the resulting behavior programming architecture complies with the design goals stated in Section 3. Application examples are given to demonstrate the respective properties.

First the compliance with the main requirements is examined:

Modularity XABSL has been a very modular approach from the start, as a complex behavior is subdivided into smaller behavior modules called options.

In the new programming language adding new options is merely a matter of adding another source file. Therefore, adding new options has become even easier. It has been shown successfully through several years of application in the *GermanTeam* that the modular nature of XABSL supports the development in larger teams very well. Also complex behaviors stay maintainable through the modularity of XABSL. This has been shown through the application not only in the *GermanTeam* but also in other *RoboCup* teams, where very large and complex behaviors have been implemented.

Table 6.1: Comparison of the complexities of some robot soccer behaviors implemented in XABSL.

	number of options	average number of states per option	number of basic behaviors	robot platform
GermanTeam 2005	113	5.58	28	*Sony Aibo*
GermanTeam 2008	112	4.43	2	*Sony Aibo*
Darmstadt Dribblers 2008 [33]	63	4.61	31	custom humanoid robot
B-Human 2008 [91]	30	4	0	*Aldebaran Nao*

Table 6.2: Comparison of the complexities of behaviors implemented for the *Urban Challenge* (cf. Section 1.3).

	number of behavior modules	
		behavior control implementation
VictorTango [47]	9	hierarchical state machine implemented in *LabVIEW*
AnnieWay [38]	15	hierarchical state machine implemented in *C++*

Table 6.1 shows the complexity of some of the largest behaviors implemented in XABSL. In contrast Table 6.2 shows the size of hierarchical behaviors developed for the *Urban Challenge*. Due to the different nature of the task, the complexity of the high-level behaviors required in autonomous driving even in very successful *Urban Challenge* applications is about five to ten times smaller than those encountered in complex robot soccer applications. Therefore, such behaviors can be implemented using standard tools such as *C++* or *LabVIEW* without resorting to specific agent behavior programming languages. When the application requires complex high-level behaviors, a behavior architecture such as XABSL is required which allows to scale up to a very large number of behavior modules.

It is worth mentioning that the behavior developed for the *GermanTeam 2008* (cf. Section 6.2.1) is no longer based on the execution of basic behaviors which is possible since the introduction of concurrent behavior execution (cf. Section 4.2). Instead, primitive robot behaviors are implemented directly in XABSL using features for continuous behavior outputs. Basic behaviors are only used for calling specific less frequently used functions such as system calls for instance for rebooting the robot. This explains why there are only two basic behaviors in the *GermanTeam 2008* code.

Portability Since the XABSL engine does not contain any platform-dependent code it is portable to any other platform.

The range of possible platforms where XABSL can be applied has even increased with the latest version of XABSL as a *Java* version of the XABSL engine has been developed.

The successful porting to different platforms besides the *Sony Aibo* robot, for instance by other *RoboCup* teams using completely different hardware running different operating systems, is proof of the good portability of the XABSL architecture. Robot hardware XABSL has been ported to include four-legged robots, humanoid robots, and wheeled robots. Besides *Aperios*, the operating system of the *Sony Aibo*, XABSL has been applied on systems running various *Windows* or *Linux* operating systems. Software architectures that XABSL has been integrated into are also not limited to those based on the *GermanTeam* architecture [88]. Worth mentioning is in particular the software framework *RoboFrame* [30, 31, 81, 82] which is applied by the *RoboCup* team *Darmstadt Dribblers* [29, 32, 33] and many other robotic projects of the author's group. See Sections 6.2.3 and 6.3 for more details about applications on other platforms.

Versatility Originally XABSL was limited to a specific type of hierarchical state machines which did not include concurrent behavior execution and only produced discrete outputs. As described in Section 4 some of the recent extensions of XABSL were aimed at lifting these limitations in order to greatly increase the versatility of the architecture.

Application examples that substantiate the increased versatility are given in the following sections.

In one example, the robot soccer implementation of the *GermanTeam* makes use of all of the added features of XABSL which allowed the team to implement simpler and more elegant solutions to realize complex soccer behaviors. When XABSL was not yet able to support concurrent behavior execution, two separate XABSL engines had to be instantiated in order to control behavior for head and legs independently of one another. This is no longer required, and head and leg behavior is now integrated into the same hierarchical state machine. Many behavior procedures concerned with continuous outputs are now implemented directly in the XABSL state machine, making use of the features supporting continuous behaviors instead of having to rely on external implementations using basic behaviors. The features supporting multi-robot cooperations are applied in order to realize dynamic role assignment which also was implemented externally beforehand.

Usability One drawback of the original XABSL concept was that behavior code had to be written directly in XML. Due to this, source code was rather verbose, and without using specific editors programming in XML was laborious. Replacing the XML dialect with the new programming language as the input language made behavior development much easier and thus increased the usability of XABSL.

New tools such as the XABSL editor have been created, and others such as the documentation generation have been further improved.

The successful realization of complex applications, as described later in Sections 6.2 and 6.3, has shown that XABSL supports the rapid development of behaviors very well. Especially in a competition situation such as the *RoboCup* it is important to be able to quickly make modifications and adaptations to an existing complex behavior. This makes the availability of good development tools a requirement.

In the remainder of this section it is shown that the extensions described in Chapter 4 can be utilized in order to realize new applications or to simplify the implementation of existing applications and thus make a contribution to meeting the stated design goals. This is done by means of application examples.

6.1.1 Concurrent Behavior Execution

The availability of concurrent behavior execution allows for many new applications that are otherwise not directly implementable. Whenever a task consists of independent subtasks that should be executed in parallel, concurrent execution is required. In previous XABSL versions this had to be realized by instantiating more than one XABSL engine and thus executing multiple separate XABSL agents in parallel.

In the *GermanTeam 2008* application (cf. Section 6.2.1) the concurrency feature was used extensively.

The most prominent example of concurrent execution in four-legged robot soccer is the separate control of head and leg motion. While the robot is not performing any motions that require the full body of the robot such as kicking or blocking motions, and while the head can be moved freely, i.e. it is not being used to catch the ball under the chin of the

robot, the head and legs can be controlled independently. The leg motion is controlled in order to walk the robot to the required spot on the field, for example towards the ball or to strategic locations depending on the current situation. The head motion will control the gaze direction of the robot. Since the robot uses directed vision with the camera in the head as the main sensor, it is important to ensure that important objects such as the ball and localization landmarks appear regularly in the field of view of the camera. Different strategies for head control will be applied in different situations. For instance, when the robot is very close to the ball it might be unaffordable to look away from the ball as the robot needs to react quickly to the movement of the ball. On the other hand only looking at the ball will not provide enough input for localization in order to track the position of the robot effectively. In the *GermanTeam 2008* application the behavior consists of two mostly independent parts: The main behavior will control the movement of the robot and set a flag selecting a requested mode of head motion. The different modes correspond to different attention priorities, e.g. specifying that the robot should only track the ball, or conversely that the robot does not need to look for the ball at all. The other part of the behavior is then responsible for controlling the head motion in accordance with the request from the main behavior.

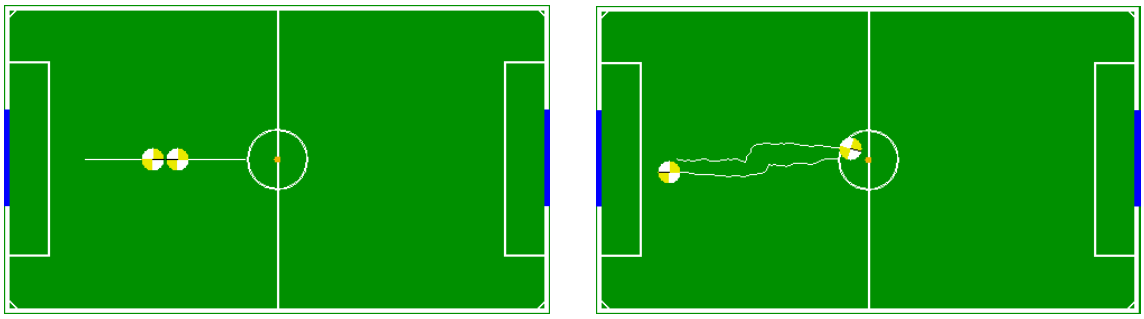
Other examples of concurrent behavior execution from the same application include supportive behaviors that should be active in parallel at all times. For instance, one option will monitor whether the sensors report that the robot has fallen over and execute a motion in order to bring the robot back onto its legs immediately, while not triggering any action otherwise. Other behaviors executed concurrently during the operation of the robot might perform simple debug outputs displaying information such as player number or team color using the LED display of the robot.

Using concurrent execution it is easy to implement features which make behavior robust. Error recovery can be implemented, for example, through an option which monitors the status of the agent and becomes active when a certain situation occurs. Such an error situation might either be detected using the external input to the agent, e.g. sensor data as in the above example, or a timeout can be implemented in order to detect that a certain subtask is not being executed correctly.

6.1.2 Integration of Continuous Behavior Control

The superposition of continuous behaviors is one of the applications which requires a behavior architecture which is able to deal with continuous behavior aspects. The continuous outputs of different behavior modules are combined, for instance by a weighted vector summation. In many applications this method is very well suited in order to assemble complex behaviors out of primitive basic behavior elements. For instance, when the output of the primitive continuous behaviors is the motion vector of a mobile robot, summation of different motion behaviors can result in a meaningful combined complex behavior. Figure 6.1 illustrates such an example. The given example was implemented in the extended XABSL architecture easily making use of the new features described in Section 4.3.

Another common case where the continuous nature of certain behaviors is utilized is the use of artificial potential fields for robot motion planning. A behavior architecture based on potential fields which was developed by members of the *GermanTeam* is pre-



(a) An application example including two robots in the *SoccerBots* [10] simulator. The robot on the left is trying to move to the center of the field, while the robot on the right is supposed to move to the left side of the field. Both robots are simply moving in a straight line towards their destination and get stuck to each other halfway.

(b) The robots are controlled by a superposition of continuous behaviors allowing them to reach their destinations successfully. One of the behavior components is moving towards the destination, another is moving away from close robots in order to avoid collisions, and the third component introduces randomness by adding some noise to the robot's movement. Without noise the robots could still get stuck to each other if they are unable to decide on which side to pass each other.

Figure 6.1: Example of continuous behavior superposition.

sented in [61]. This architecture has been applied in combination with XABSL, where low level potential field behaviors have been implemented using basic behaviors [92]. In the extended XABSL architecture the integration of continuous behaviors such as artificial potential fields has become more flexible through the new features supporting continuous behavior aspects. Externally generated continuous variables can be interfaced as input symbols. This allows for more transparent integration than simply transferring control to an external basic behavior.

In four-legged robot soccer another case where continuous behavior aspects could easily be integrated can be found in the example of estimating the time a robot will need to get to the ball. In the *GermanTeam 2008* robot soccer implementation such an estimation is required for the dynamic role assignment (cf. Section 6.2.1.1). While the estimation is a continuous variable which gets calculated from values such as the current distance to the ball and the time since the ball was last seen, it also depends on the discrete state of the robot. For instance, when the behavior is in a state where it is clear that the ball is currently grasped under the chin of the robot the estimated time will be zero independent from current observations. Therefore, the estimation of the time until the ball can be reached is performed inside of the hierarchical finite state machine in a specific option which is executed concurrently as long as any behavior is executed which requires role assignment. One of the factors considered when estimating the time to reach the ball is an approximation of the anticipated time lost while approaching the ball due to collisions with other robots. The sensory inputs which suggest possible collisions consist of binary flags for detected obstacles in the proximity of the robot or in the direction of the ball. It is assumed that the time required to get out of collisions is large when there were many close obstacles detected during the last frames. As discontinuities in the estimation should be prevented in order to facilitate a stable role assignment, the estimated time lost

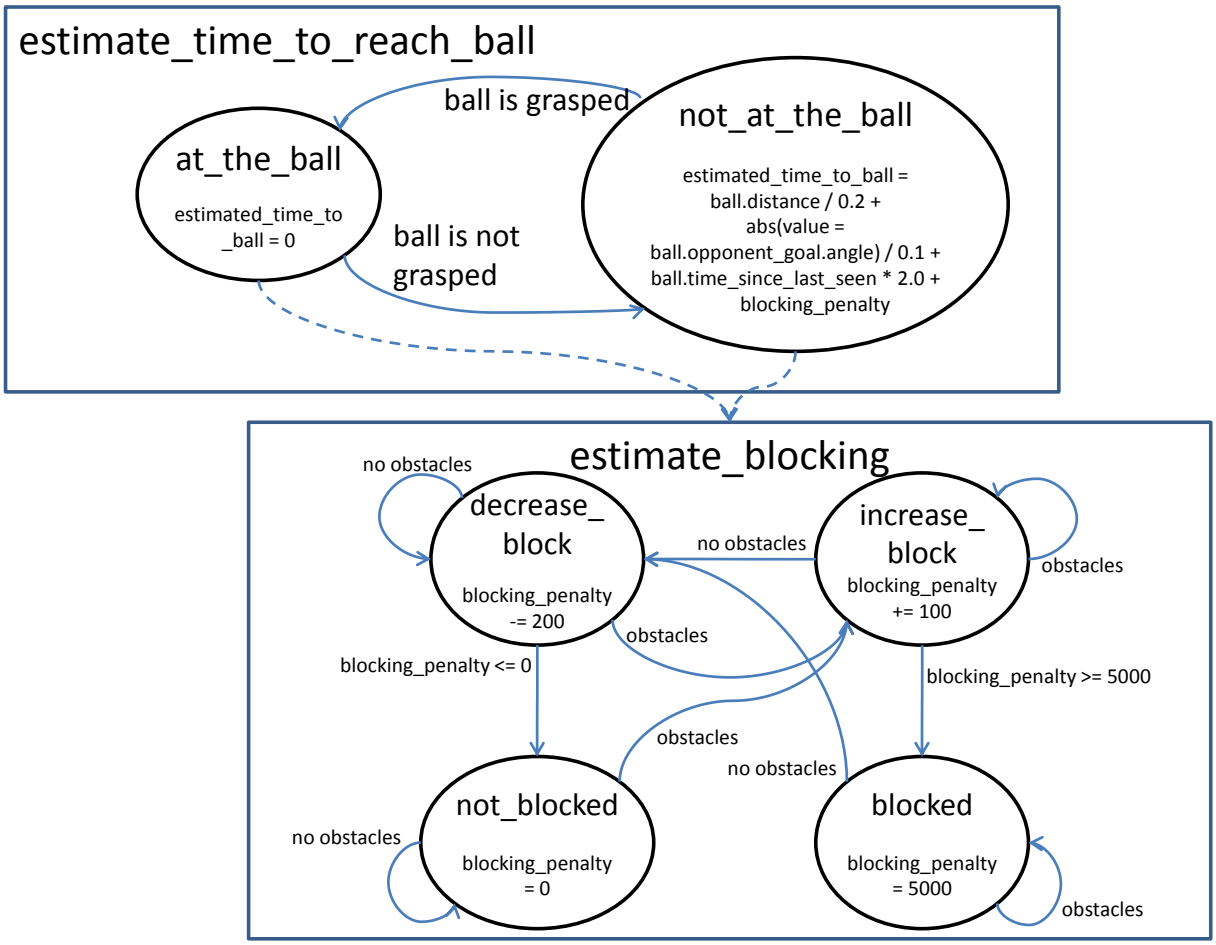


Figure 6.2: An example from the *GermanTeam 2008* robot soccer implementation for calculating a continuous estimation of the time until the ball can be reached.

due to collisions should also change only gradually. Therefore, the applied strategy for estimating the time lost due to collisions, is to increase the estimation when obstacles are detected, and to decrease it when there are no obstacles. This is also done by a separate option of the hierarchical state machine. Figure 6.2 depicts the part of the hierarchical state machine which performs these estimations.

6.1.3 Cooperative Multi-Robot Systems

The most obvious example of multi-robot cooperation in the robot soccer scenario is the dynamic assignment of player roles. According to the current situation teammates have to coordinate with each other constantly in order to perform any meaningful team play.

In particular, it is important to decide which of the field players is supposed to attack the ball. In general the best strategy is that only one of the players will head for the ball while the other players perform different duties. When more than one player on the same team attacks the ball, they end up hindering one another. The player who will attack the ball, called the striker, is typically the one who is closest to the ball. For more details on the role assignment see Section 6.2.1.1.

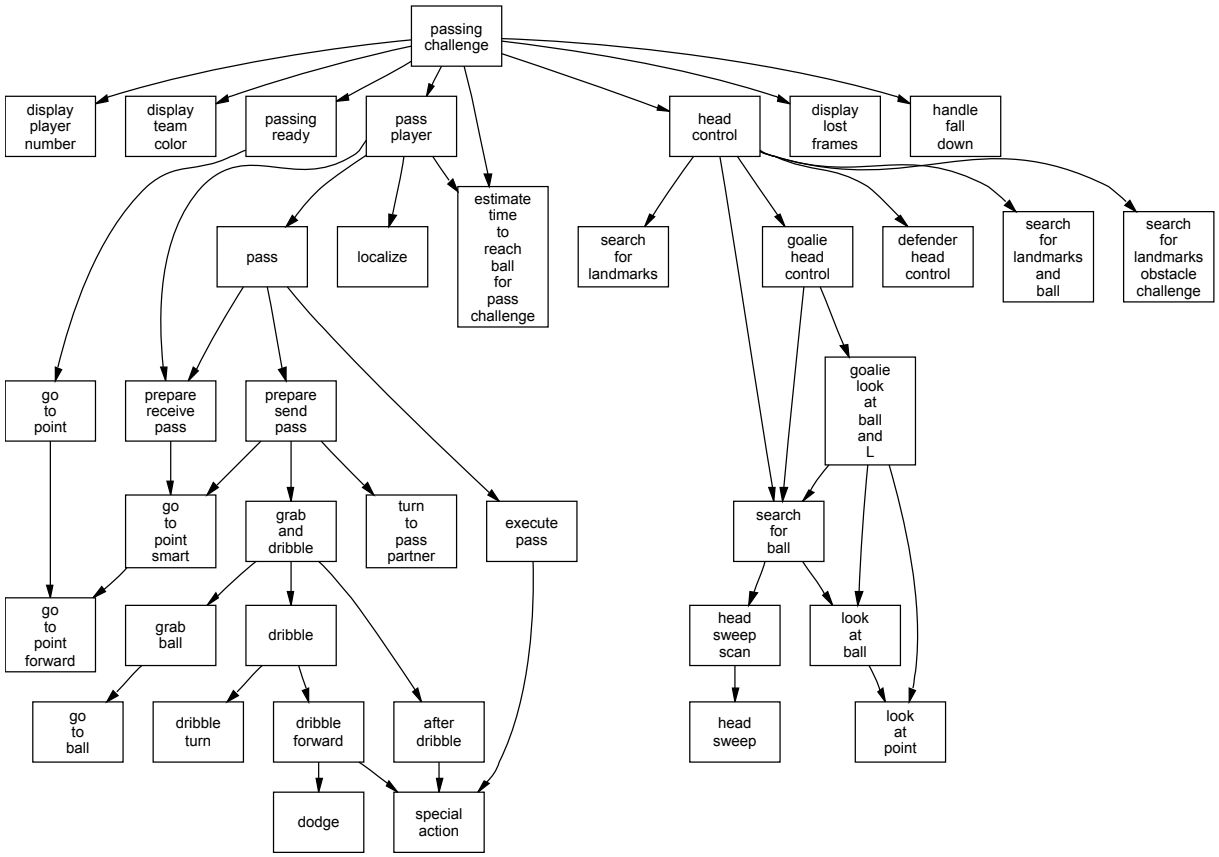


Figure 6.3: The option graph of the Passing Challenge example

On the one hand it is important that during the whole game there is exactly one player who is attacking the ball. If in a situation there are two field players assuming the striker role, this can have a negative impact on the course of the match, and if no player attacks the ball, this would be even worse. On the other hand it is also very important that the decision is made as quickly as possible. There must not be any hesitation when deciding which player will assume what role. The assignment must also be stable. Small changes in the current situation must not lead to oscillations in role assignment. Finding an implementation that meets these requirements is not trivial. The capacity state feature of XABSL described in Section 4.4 can be applied in order to guarantee that only one robot will assume the striker role (cf. Fig. 4.7(a)).

Another example of a multi-robot application realized using the new cooperation features was successfully implemented. The scenario of the application is the 2007 Passing Challenge of the *RoboCup Four-Legged League*. Three *Sony Aibo* robots are supposed to pass an orange ball to one another (cf. Fig. 6.4). In this example both presented features for the specification of cooperative behaviors are used. Utilizing a capacity state a task assignment is realized similarly to the previous example in order to decide which of the robots will go to the ball and catch it while the others wait until they receive a pass. When performing the pass both robots synchronize their actions as described in the previous example (cf. Fig. 4.7(b)). Fig. 6.3 shows the option graph used for this application. This implementation also is a good example for the support of code reuse as

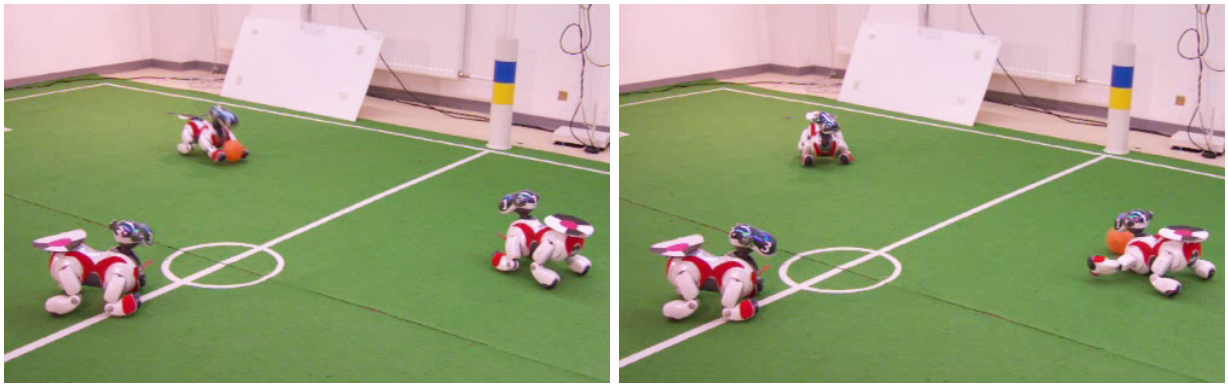


Figure 6.4: A successful pass between two four-legged robots (cf. Figs. 4.7, 6.3)

most of the required options, such as behaviors for controlling the ball, could be taken from the standard robot soccer application.

6.1.4 Machine Learning and Optimization

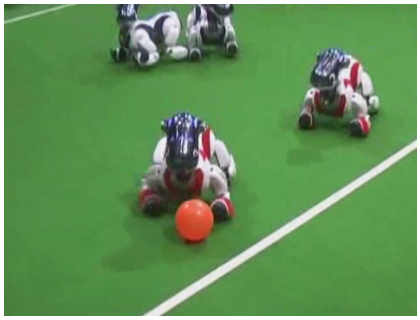
In the following section some examples are presented that demonstrate the possibility of combining behavior programming with hierarchical state machines and automatic behavior generation methods, such as optimization and machine learning.

Grab Optimization As mentioned in Section 4.5 the ball grasping (or grab) task in the *RoboCup Four-Legged league* has been addressed using *Asynchronous Parallel Pattern Search* [42, 57].

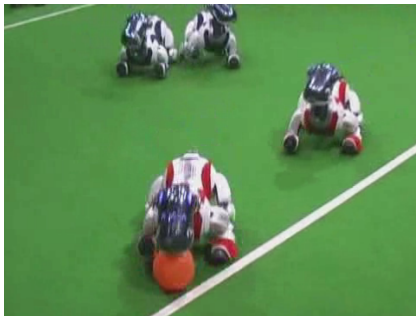
The ball grasping behavior resides in one of the options of the XABSL hierarchy. Therefore, testing and optimizing this sub-behavior can be done independently from any other implemented behavior options. The resulting optimal ball grasping behavior can be applied by different options in different behavior contexts. The hierarchical nature of the behavior architecture allows optimizing subtasks separately.

The optimization problem in this case is to determine the continuous parameters of a given ball grasping behavior. The parameters can be values such as distance thresholds, execution times, and motion speeds. For each parameter set a test run consisting of five independent trials is conducted. As evaluation function the average time until the ball is grasped is used, where every unsuccessful trial in which the ball was not securely grasped is accounted for by a very high penalty value. In each trial the ball is placed at a random location on the regular playing field and the robot positions itself 50 centimeters away from the ball. Then the grasping behavior is triggered and the time taken until completion is observed. Success of the behavior is checked by letting the robot turn on the spot. Only if the ball is not lost while rotating, the ball was securely grasped and the trial deemed successful.

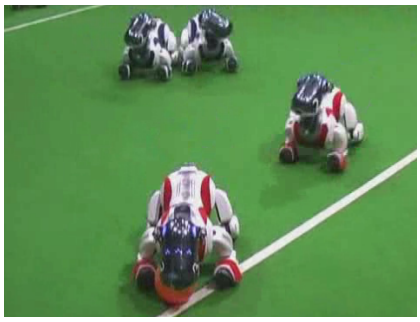
The optimization can be run semi-automatically with only minimal human intervention. As the robot positions itself automatically before starting each trial no manual repositioning of the robot or the ball is required. Only if the ball has left the playing field, which will happen occasionally, it must be replaced at the center of the field, and



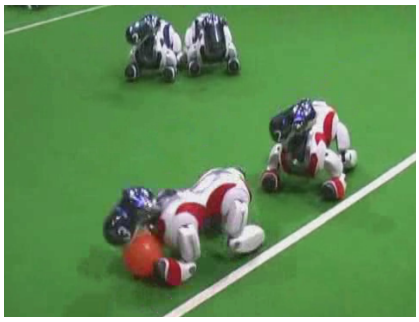
(a) The robot moves towards the ball,



(b) until the ball is in the correct position,



(c) lowers its head,



(d) and is able to move the ball into the desired direction.

Figure 6.5: A four-legged robot successfully grasping the ball during a soccer match.

approximately twice per hour the battery of the robot needs to be replaced. Besides that no intervention is required during optimizations runs.

The last optimization run resulted in the behavior which was employed by the *GermanTeam* at *RoboCup* 2007 and 2008. The set of parameters was reduced to only two continuous parameters, namely the distance to the ball at which the grab is triggered and the time required to secure the ball underneath the chin. Because of the small state space near optimal values could be determined by a small number of evaluations. The initial parameter set was a manually tuned solution with which the robot was able to successfully grasp the ball in 7 out of 10 trials with an average time of 1.6 seconds per trial. After 16 evaluations the best parameter set was found. It allows the robot to securely grasp the ball in every trial requiring 1.49 seconds on average. Figure 6.6 shows the progress of the optimization run.

Images of the resulting behavior are shown in Figure 6.5.

Intercept Learning In order to illustrate the ability to easily combine machine learning with behavior specified in hierarchical state machines, we implemented a small example in the *SoccerBots* [10] simulation. In this example the learning task was ball interception. The goal in this scenario is for the single simulated robot to reach the ball as quickly as possible. The obvious strategy of moving directly towards the ball is only optimal when the ball is motionless. If the ball is moving, the optimal intercept behavior needs to anticipate where the ball is going.

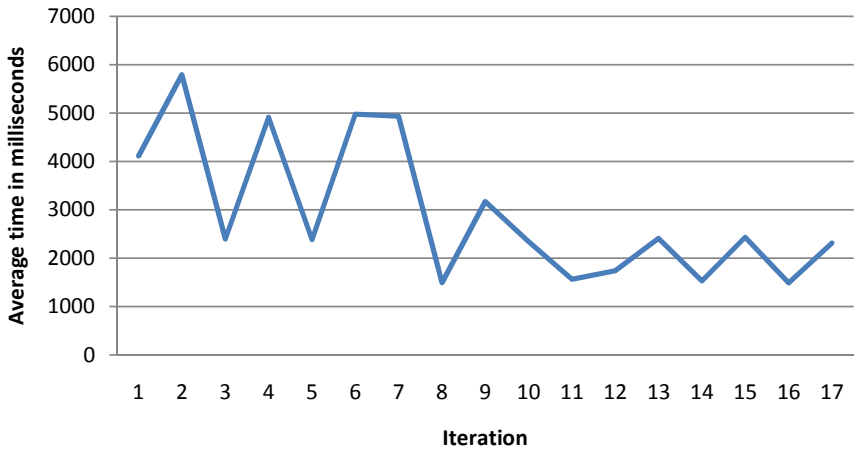


Figure 6.6: The progress during ball grasping optimization. The curve shows the average time per grasping trial averaged over 10 trials. Each unsuccessful grasping trial is accounted for with a penalty value of 10 seconds.

The task is modeled as a reinforcement learning problem with three discrete actions: turning left, turning right, and moving straight ahead. Thus, the robot is always either turning on the spot or moving forward with a constant speed. There are four continuous state variables: the direction to the ball, the distance to the ball, and the two components of the velocity vector of the ball. Learning is episodic where in each episode at the start, the robot and ball are placed at fixed positions while the orientation of the robot and the initial velocity of the ball are selected randomly. This ensures good coverage of the state space. Only the distance to the ball is kept constant at each episode start. This will, however, be reduced over the course of an episode. An episode ends when the robot reaches the ball. The reward in this problem is positive when the robot is moving closer to the ball and negative when it is moving away from the ball.

The learning algorithm applied is linear, gradient-descent *Watkins* $Q(\lambda)$ with binary features, ϵ-greedy policy, and accumulating traces [98] (cf. Section 2.11.1). We are using ten $32 \times 4 \times 4 \times 4$ hyperrectangular overlapping gridtilings, offset by random fractions of the tile width. Parameters values used were $\lambda = 0.9$, $\gamma = 0.95$, $\alpha = 0.004$, and $\epsilon = 0.3$.

Figure 6.7 shows the average number of steps per episode during learning. It can be seen that a good policy is learned within 1000 episodes. Following the best policy found by learning, the ball is reached on average in 50 simulation steps. Figures 6.8 and 6.9 give a qualitative impression of the resulting learned behavior, the first one showing examples of the behavior early during learning where a good policy has not yet been established, while the second shows resulting robot and ball trajectories when the learned policy is applied.

Similar learning problems have already been addressed successfully with reinforcement learning (e.g. cf. [104]). Therefore, the novelty of this application lies not so much in the successfully learned behavior, but rather in the smooth integration of a learned subtask into a more complex behavior realized with hierarchical state machines. In fact, the learned intercept behavior is part of a soccer demo behavior. In order to execute the learning run described above, nothing else was required than to modify the agent so that it would only trigger intercept behavior, and to modify the simulation so that it would

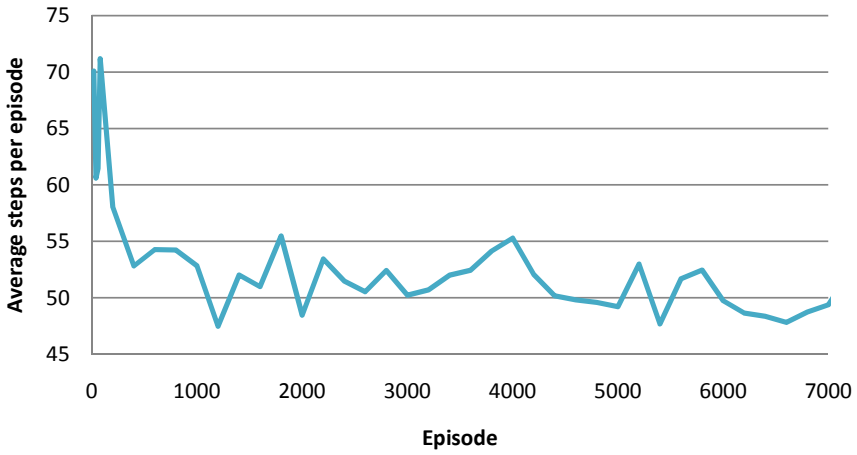


Figure 6.7: The learning curve for the ball interception reinforcement learning. The curve shows the number of steps per episode averaged over 100 episodes.

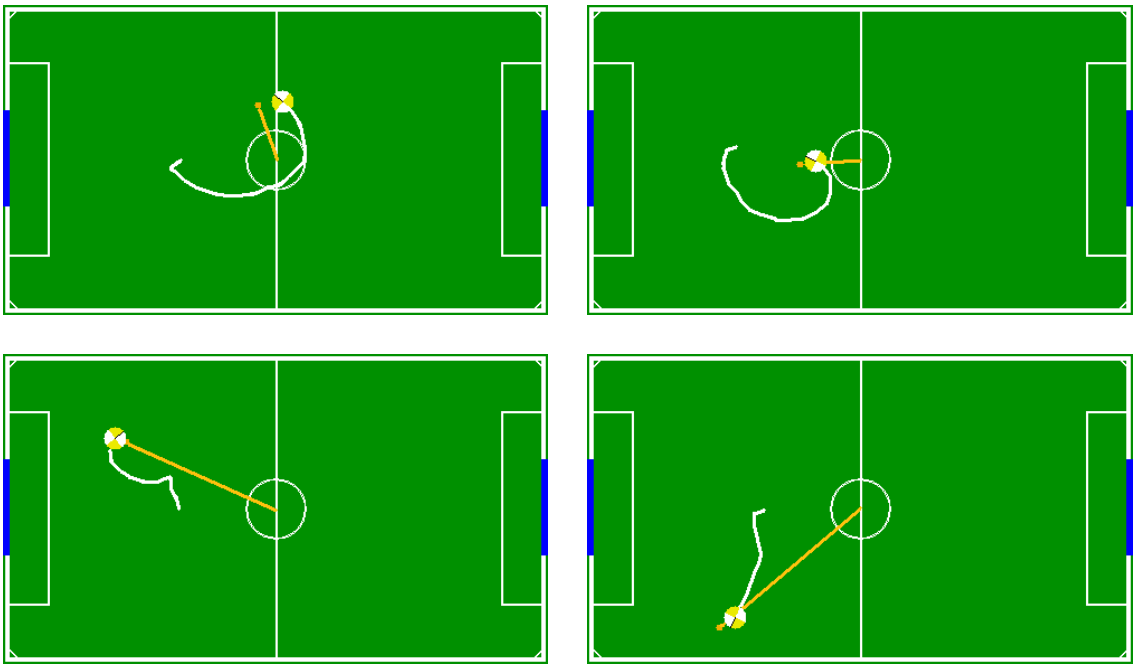


Figure 6.8: Examples of ball interception using a poor policy just after learning has started.

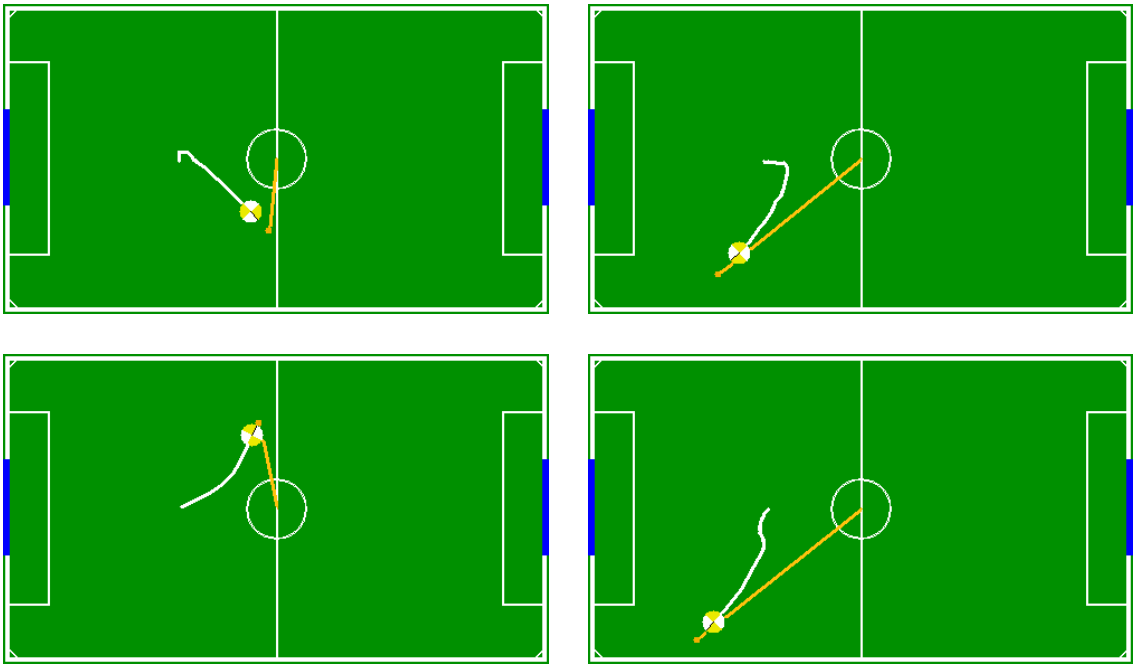


Figure 6.9: Examples of ball interception following a learned policy.

reset the scene every time the robot reaches the ball. It is even possible to enable learning during a soccer match, although it would be required to adjust the learning problem since it has to be considered that the ball is being moved by other robots. Figure 6.10 shows how the intercept behavior is integrated into the XABSL state machine.

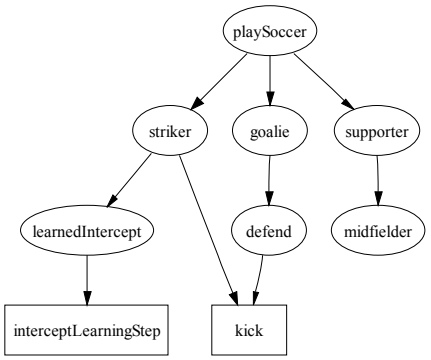
6.2 Applications in RoboCup Soccer

The robot soccer application developed by the *GermanTeam* is not only the main test application for XABSL, it is also one of the most complex applications that has been realized with XABSL up to date. It is presented in this section in some detail. Furthermore, similar applications by other teams in the *Four-Legged League* and also by teams in different leagues are discussed.

Robot soccer is well suited for evaluating behavior control solutions, since it is a complex real-world task which also requires a very complex behavior control. Other real-world applications that also have a very large complexity often only exhibit relatively simple behavior control solutions. For instance many applications in autonomous driving can be solved with rather simple behavior control even if the task is very complex (cf. Section 6.1). One aspect responsible for the behavior control complexity of robot soccer is the large variety of possible behavior choices each robot has in any game situation. Each player has to decide whether it will try to acquire the ball or moves to a strategic position. If a robot reaches the ball it has to decide in which direction and by what means it will play the ball. Players have to interact with each other in order to achieve cooperative team play. Therefore decision making in this scenario requires very complex solutions.

```
option learnedIntercept
{
  initial state executePolicy
  {
    action
    {
      interceptLearningStep();
      steerHeading =
        (getBestInterceptAction == turnLeft) ?
          steerHeading + 0.1 :
        ((getBestInterceptAction == turnRight) ?
          steerHeading - 0.1 :
          steerHeading);
      speed = (getBestInterceptAction == goStraight) ?
        1 : 0;
    }
  }
}
```

(a) Integration of the intercept learning behavior shown by the XABSL source code of the option *learnedIntercept* which executes a learned intercept policy. The input symbol *getBestInterceptAction* returns the action to be executed according to the current policy. The basic behavior *interceptLearningStep* is called in each cycle and can perform any required steps such as the actual learning.



(b) The option graph of the soccer demo including the option *learnedIntercept* and the basic behavior *interceptLearningStep*.

Figure 6.10: Integration of the intercept behavior in the hierarchical state machine.

6.2.1 GermanTeam 2008

Since being developed by members of the *GermanTeam* starting in 2002, XABSL has been in use at *RoboCup* competitions every year and has helped the team become world champions three times in 2004, 2005, and 2008. In addition to the international competitions, the cooperation partners of the joint project *GermanTeam* were all employing XABSL when participating separately in national competitions at *RoboCup* German Open.

Since the *GermanTeam* is a cooperation from three (formerly four) universities, many people were involved in the development of the behavior of the soccer playing agent. Temporarily there were about 50 people in different locations working on the agent at the same time. The modular approach of XABSL supports distributed development in large teams very well. New behavior options can be added easily without necessarily having an effect on existing options and options can be tested independently from one another.

Of course the decision making module is only one of the parts required to be successful in robot soccer. Other important parts include image processing, world modeling, and motion generation. In the Four-Legged League specific difficulties arise due to the relatively high complexity of the motion capabilities (compared to leagues using wheeled robots for instance) and directed vision requires coordination of vision and motion as well as robust world modeling which is able to integrate noisy perceptions of the partially observable environment into one consistent model. More information on the *GermanTeam* software architecture and the implemented modules used in 2008 and previous years can be found in [11, 87, 88, 92]. A complete list of publications of the *GermanTeam* can be found online [86].

Behavior control takes a special position among the required modules as the possible behavior strategies depend very strongly on the quality and the performance of all the other parts of the robot's software system (cf. Fig. 6.11 and Fig. 6.12). For instance, advanced behavior strategies that include all of the positions of robots of the opponent team are feasible only when there is a robust detection and tracking of the opponent robots providing the required information (which usually is not the case in the Four-Legged League). In another example, the decision whether or not to pass the ball to a teammate depends on several criteria such as the physical ability to kick the ball at an exact angle and to safely receive an incoming pass, the ability to recognize the direction towards the pass partner, and also the speed of the robot while playing the ball and when walking without the ball.

Thus, it is obvious that the behavior control is responsible for making the best use of the available information provided by the other modules, so that the resulting behavior is as efficient as possible. It is important that, especially before or at competitions, behavior developers are able to quickly react to changed premises, for example, when one of the modeling modules has been improved in a way which opens new strategic behavior options. The modular nature of XABSL supports this kind of rapid development very well.

During the years of development the *GermanTeam* has developed a library of well-tuned and exhaustively tested behavior options which can be reused in different contexts in order to quickly develop new behavior options.

In this section the robot soccer playing behavior used by the *GermanTeam* at *RoboCup* 2008 is described in more detail.

The soccer behavior is split up in 66 behavior options. The number of options is relatively small compared to some of the behaviors implemented in previous years. The number of options was reduced, for instance, by merging some options of similar functionality into a single option where the exact function is determined through option parameters. Thus, redundant options were avoided and the maintainability was improved. The option graph of the soccer behavior is shown in Figures 6.13, 6.14. Basic behaviors are not used. The complete behavior implemented by the *GermanTeam* consists of 112 options. Besides the actual soccer behavior implementation it also contains behavior for technical challenges and various demos and tests.

Some of the options are responsible for long-term, deliberative decisions such as selecting the current role of a robot. These options usually are positioned at the top levels of the option graph. Other options can also contain very reactive behaviors which have to respond quickly to changes in the environment of the robot, for example for looking at or going to the ball. The latter are usually found in the lower option levels. The hierarchical structure of the behavior supports combining different behavior types on different hierarchy levels.

6.2.1.1 Role Assignment

Since a team of autonomous agents has to be coordinated, the task of playing robot soccer is highly cooperative. The general strategy in robot soccer usually is to have only one of the players try to control the ball. If more than one robot walks up to the ball and tries to kick it, they will get in one another's way. Therefore, the team must first

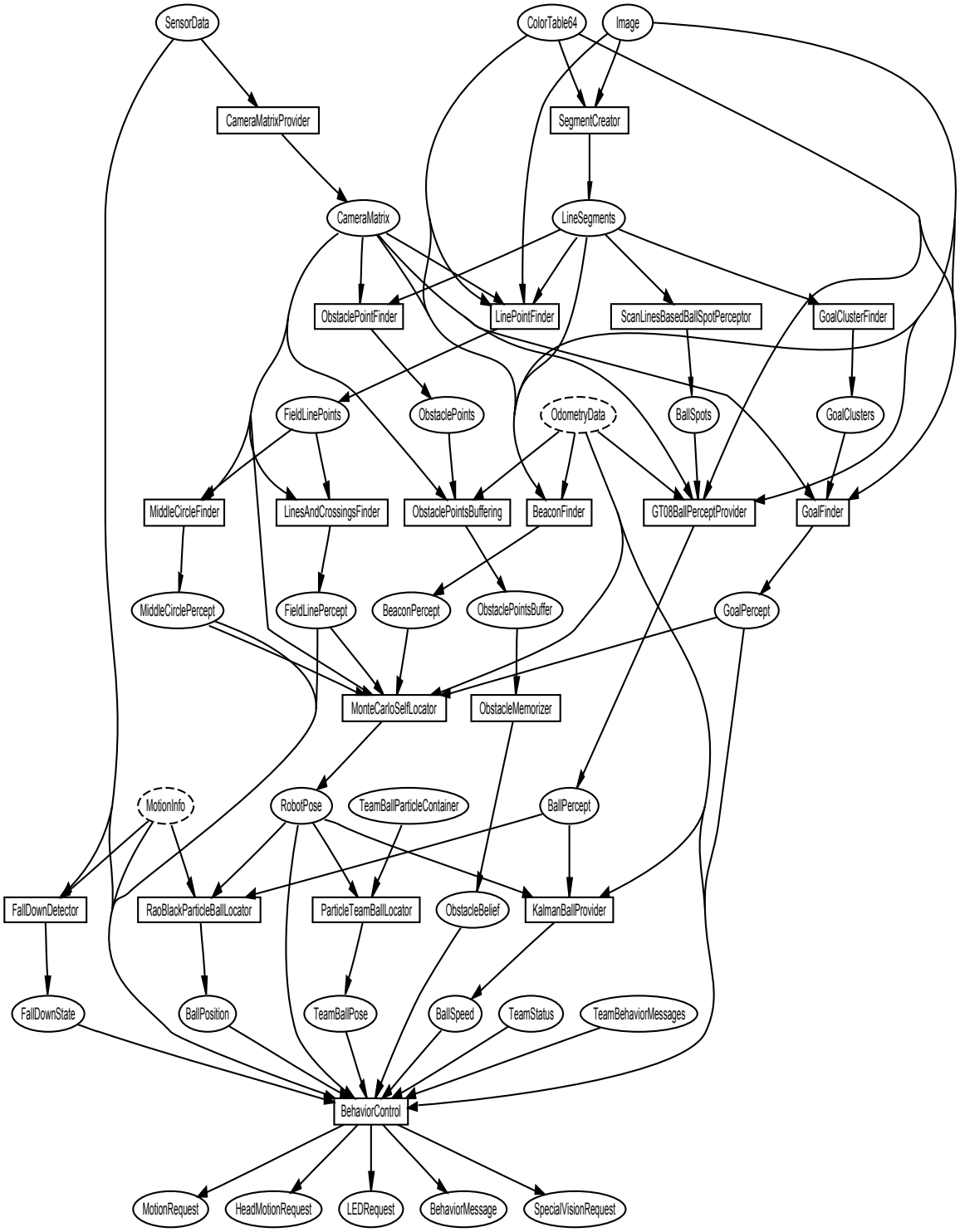


Figure 6.11: Information flow in the *cognition* process of the *GermanTeam* 2008 software. Boxes represent software modules, while ellipses are data representations exchanged between the modules. This is a partial view showing the most important modules only. The module *BehaviorControl* contains the decision making component realized with XABSL. It collects all necessary information from the other modules on which decisions are based on and generates the representations required as inputs for the motion generation process.

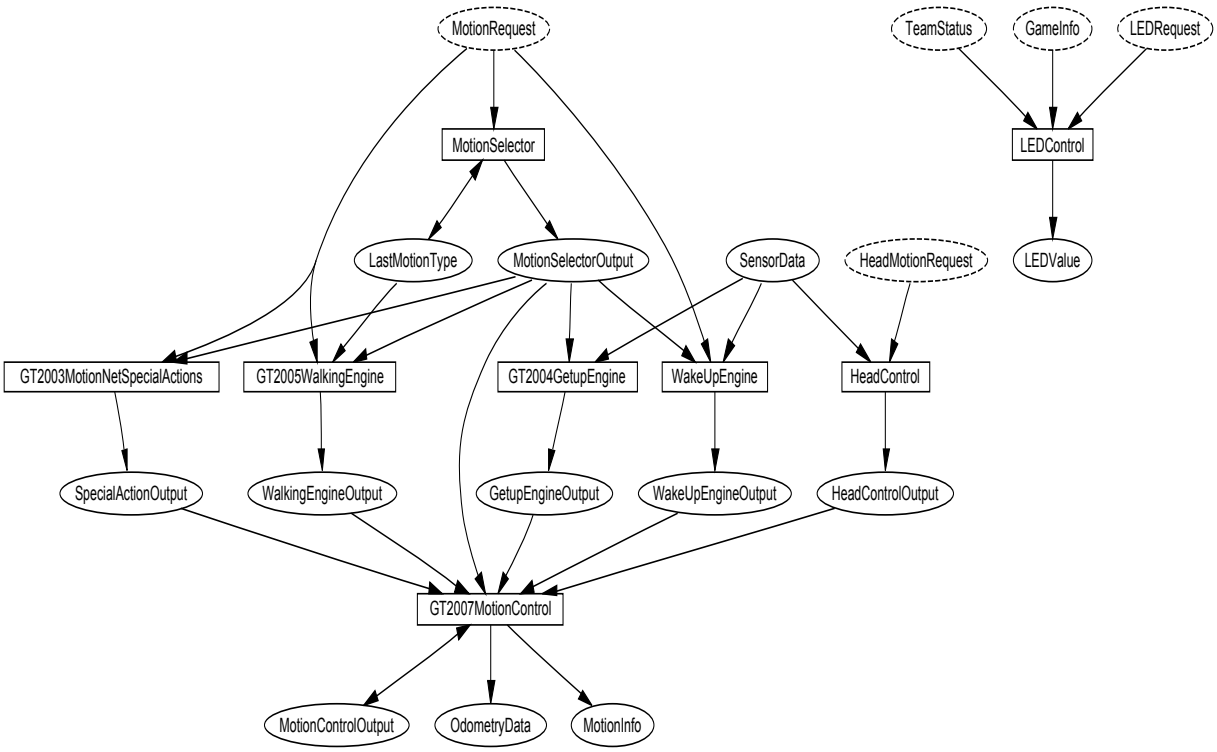


Figure 6.12: Information flow in the *motion* process of the *GermanTeam* 2008 software.

decide dynamically which player will attack the ball and assume the so-called *striker* role. Usually this should be the player who currently is closest to the ball. After the striker has been determined, the other field players have to position themselves on the field strategically assuming different supporting positions. According to the rules of the Four-Legged League the robot with the player number one is the only one allowed in front of the own goal and therefore will always assume the goalie role and does not take part in the dynamic role assignment.

Deciding which of the players is closest to the ball is not as trivial as it may seem at first. There is no external instance which will perform a centralized role assignment. Each robot has to decide on its own which role it will assume. Still it is necessary that the team of robots come to consistent decisions, for instance, it should be guaranteed that there are not two robots assuming the striker role simultaneously for a significant period of time. The decision have to be made based on noisy observations. That means there are, for instance, no exact measurements of the distances from each robot to the ball. In addition, while the robots are able to communicate via wireless network, the communication does not occur spontaneously. Therefore, network delays might also cause conflicts in role assignments.

For dynamic assignment of the striker role each robot will estimate the time it would require to get control of the ball. This estimation includes the last perceived distance to the ball, but also the last time when the ball was seen and the angle to the opponent goal. A robot that is already aligned towards the opponent goal will receive a lower estimation value and will thus more likely assume the striker role. Another factor accounted for in the estimation is a value reflecting whether or not the robot has detected obstacles in its direct

vicinity and how much time has passed since close obstacles were detected. A robot which is currently or was recently blocked by obstacles will receive a penalty to its estimated time to reach the ball. If a robot is already in control of the ball it has an estimation value of zero. Negative estimation values are not allowed (cf. Section 6.1.2). The robots exchange their respective estimated values, and the one with the lowest estimation assumes the striker role. The synchronization features of the XABSL engine (cf. Section 4.4) are used to ensure that only one robot at a time will become striker.

Since each team consists of five players besides the goalie and striker, there are three supporter roles to be assigned. The three supporter roles are offensive supporter, defensive supporter, and defender. They will position themselves on different strategic points on the field. The three roles are assigned according to the current positions of the players.

6.2.1.2 Ball Handling

The robot that has been assigned the striker role is responsible for attacking the ball and playing it towards the opponent goal. Except in situations where the ball can be played in the right direction by executing one of its kicking motions, the robot will try to grab the ball under its head, thus getting exclusive control of the ball. According to the rules this is considered holding the ball and is only allowed for at most three seconds. The amount of time the robot is holding the ball can easily be monitored by watching the activation time of the option which is active while the ball is grabbed (which is called *dribble*) and aborting it – and thus all subsequently activated behaviors – when the time limit has been reached. While holding the ball the robot will try to turn towards the opponent goal and then move the ball forwards until there is a good opportunity for performing a strong kick to the opponent goal. When the robot detects an obstacle in its path it will try to avoid it by moving sideways with the ball (called *dodging*). When the time limit is exceeded the ball usually is released by executing a soft kick in the robot's current direction.

6.2.1.3 Supporter Positioning

The three supporting field players are not supposed to attack the ball. Instead whenever they come closest to the ball, a role change occurs and the robot becomes the new striker. Therefore, it is enough for the supporting robots to position strategically in order to occupy relevant positions on the field. At least one of the supporters will stay close to the ball, trying to position close to the striker, while not obstructing the striker, avoiding collisions and trying not to stand in the striker's path. The offensive supporter will always stay in the opponent half of the field, while the defensive supporter stays in the own half. The defender always keeps in front of the penalty area close to the goalie. Whenever the ball is close to the field border the supporter which currently is not in the same half of the field will position next to the throw-in points at the center line of the field where the ball will be put back into play after it has been played out of the field by one of the robots. Figure 6.15 shows examples of player positions from real game situations from the final match of *RoboCup* 2008.

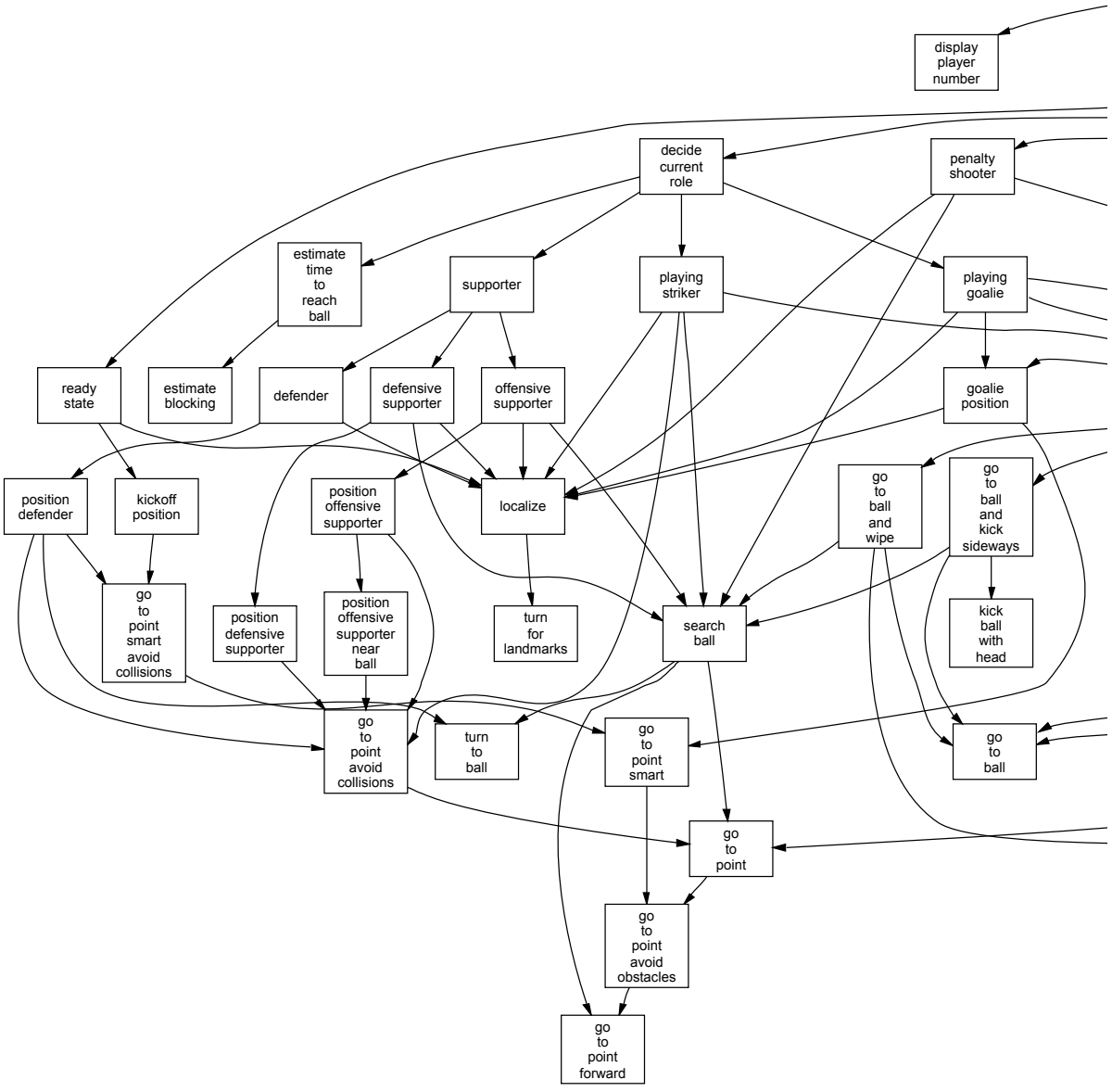


Figure 6.13: The option graph of the *GermanTeam 2008* soccer playing behavior agent (part 1).

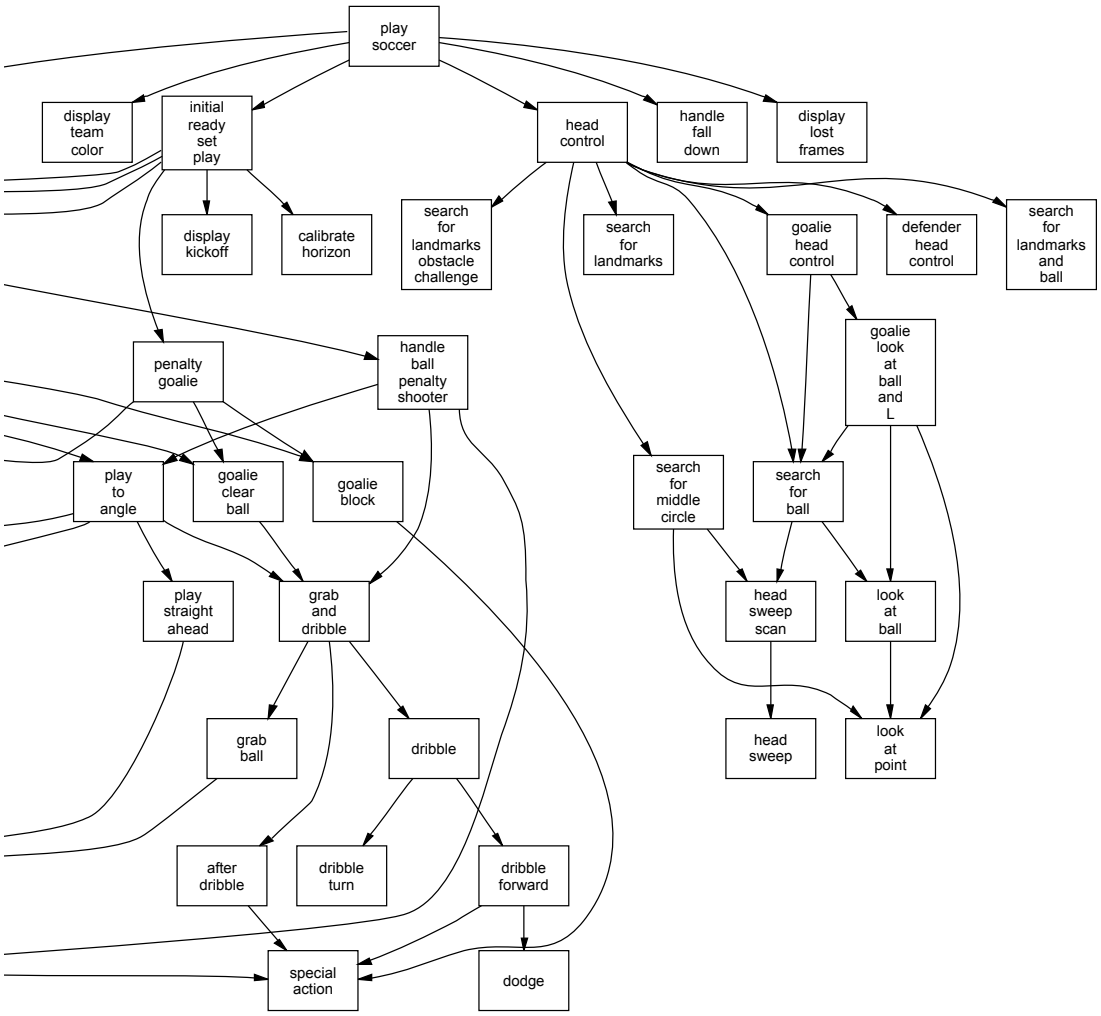
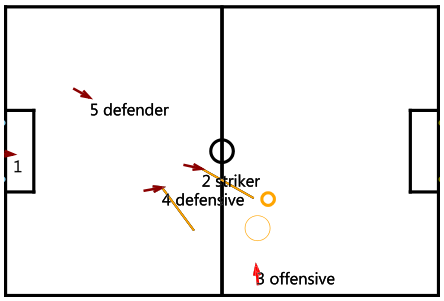
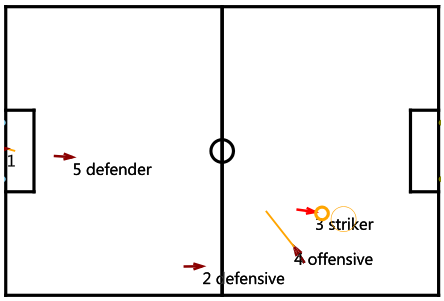


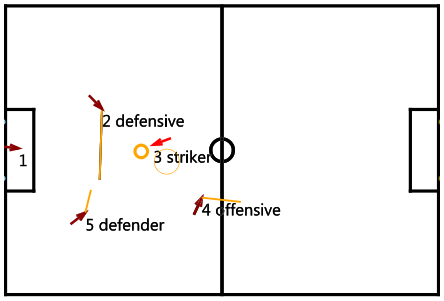
Figure 6.14: The option graph of the *GermanTeam 2008* soccer playing behavior agent (part 2).



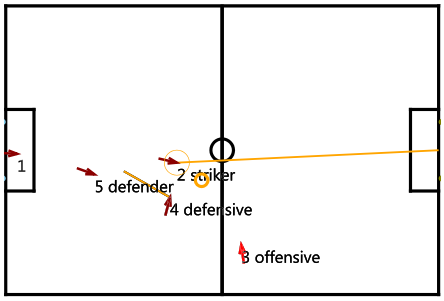
(a) Ball in opponent half.



(b) Ball in opponent half. Defensive supporter moved to throw-in point.



(c) Ball in own half.



(d) Ball in own half. Offensive supporter moved to throw-in point.

Figure 6.15: Examples of player positioning

(a) Goalie positions at the goal,

(b) blocks a shot,

(c) clears the ball,

(d) and returns to the goal.

Figure 6.16: Scenes from a four-legged robot soccer match demonstrating the goalie behaviors.

6.2.1.4 Goalie Behavior

The goalie is the only player that is allowed to stay inside of the penalty area. Therefore it stays inside the penalty area all of the time. It will attack the ball when it is inside or very close to the penalty area but will return to the goal immediately when the ball is no longer dangerously close to the goal. The goalie will position close to the goal between goal and the ball. If the goalie does not see the ball it stays in the center of the goal. While positioning, a behavior option is executed concurrently which can execute a blocking motion whenever the ball seems to roll towards the goal and the speed of the ball is too high to safely intercept the ball without a blocking motion. Figure 6.16 shows some pictures from a typical game situation demonstrating the goalie behavior.

6.2.1.5 Head Control

As the robots are using directed vision having only one camera mounted on their heads, controlling the movement of the head is an important part of the robot's behavior. The behavior for control of the head is implemented in a part of the hierarchical state machine which is executed concurrently to the other behaviors. Other behavior options can request specific modes of head movements. The most important mode is searching for the ball and tracking it when it has been detected in the camera image. But as the robot also has to localize on the field and detect the other relevant objects using perceptions from the camera, only tracking the ball is not a sufficient strategy. Therefore, there are other head control modes where the head will search for landmarks on the field or track the ball most of the time and search for landmarks occasionally.

6.2.2 Other Teams in the Four-Legged League

Since XABSL has been applied by the *GermanTeam* for many years with remarkable success there has been considerable impact, at least in the *Four-Legged League* community. The complete software and documentation developed by the *GermanTeam* was published occasionally and is available online from [86]. Many teams have used the code releases as a starting point for their own developments. As XABSL is contained in the code releases, these teams are also using XABSL for behavior control. Here is a list of teams that have been applying XABSL at competitions in the *Four-Legged League*:

- *Microsoft Hellhounds* was a team from *Universität Dortmund* made up of former members of the *GermanTeam*. Similarly to the approach described in Section 6.1.4 they also implemented learning of ball grasping parameters [45].

- *Hamburg Dog Bots* started in 2005 using *GermanTeam* code from 2004 including XABSL, which they applied until it was replaced with their own behavior scripting language *CL2* in 2006 [56].

- *WrightEagle* also used the *GermanTeam* 2004 code as a basis for their own developments. They developed an editor for XABSL documents [106].

Figure 6.17: Team *CoPS* in the *RoboCup Middle Size League*.

- *TecRams* have developed a translator from an XML behavior description to XABSL. They are using XABSL in order to execute behaviors which are partly generated using Evolutionary Programming [99].
- *sharPKUngfu* also based their development on *GermanTeam* 2004 code and thus use XABSL as well [102].
- *Harzer Rollers* are a German team which has only participated at *RoboCup German Open*. They are specifying behaviors using hybrid automata (cf. Section 2.9) and have developed a translator to produce XABSL code which they can execute on their robots [93].
- *Dutch Aibo Team* is a joint Dutch team who was applying XABSL at their *RoboCup* participations between 2004 and 2006 [80],[101].

This list may be incomplete, as teams are not required to publish their source code after competitions. Therefore, it is possible that other teams are using XABSL as well. In any case, it can be observed that XABSL enjoys a great deal of popularity among the teams in the *RoboCup Four-Legged League*.

6.2.3 Teams in Other Leagues

In *RoboCup* XABSL is not only being applied by a large number of teams in the *Four-Legged Robot League*, but also on different robots by teams in all robot leagues:

In the *Middle Size Robot League* (cf. Fig. 6.17) the team *CoPS* [60] developed a graphical behavior modeling tool using Petri nets which can automatically generate XABSL source code [108].

The *Small Size Robot League* team *B-Smart* uses XABSL to control the behavior of their robots [15].

In the *Humanoid Robot League* XABSL is applied by several teams including the *Darmstadt Dribblers* [29, 32, 33]. The *Darmstadt Dribblers* have participated in the *Humanoid Robot League* since 2004 and reached the quarterfinals in 2007 and 2008. The robots used

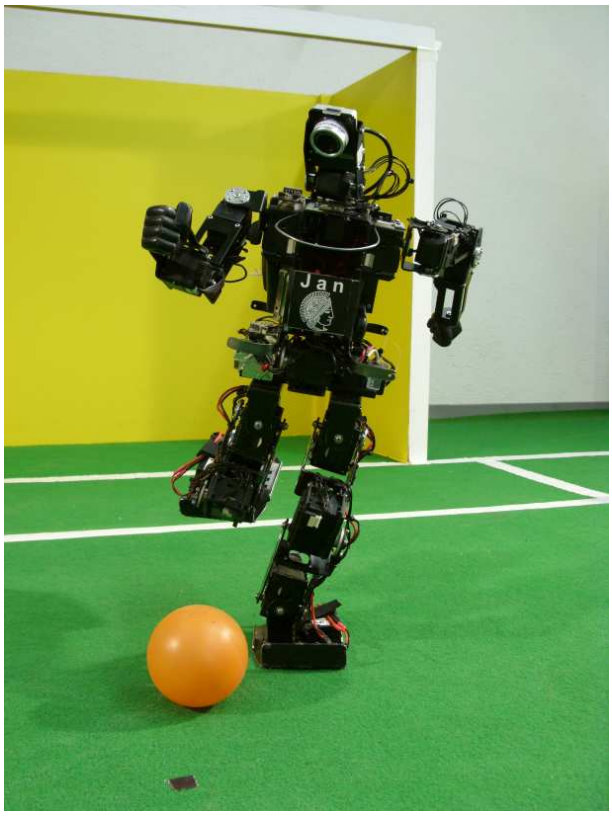


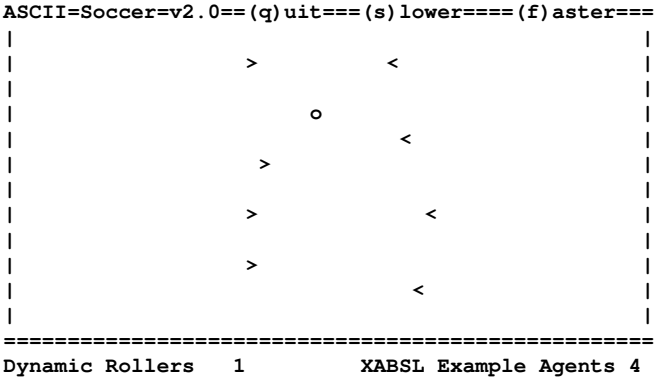
Figure 6.18: Humanoid robot of the *Darmstadt Dribblers* kicking a ball.

by the *Darmstadt Dribblers* have a kinematic structure with 21 degrees of freedom. They are equipped with an articulated camera and distributed computing hardware, consisting of a controller-board for motion-generation and stability control and an embedded PC board for all other functions. For motion stabilization three one-axis-gyroscopes and a three-axes-accelerometer are used. One of their robots is shown in Figure 6.18. For behavior control the team has been applying XABSL since 2005.

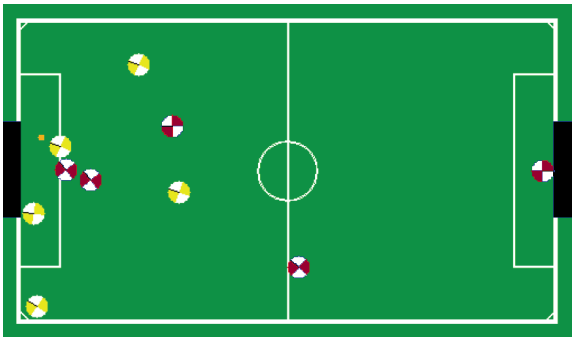
Other teams that are or have been active in the *Humanoid League* applying XABSL for behavior control are *B-Human* [62, 89], *DohBots* [24], *BreDoBrothers* [90], and *HumanoidTeamHumboldt* [46].

In the *Standard Platform League Nao Division* all of the teams are using the humanoid *Nao* robot as a standard platform. This league was founded as a replacement for the *Four-Legged League* which since 2007 is also being called *Standard Platform League Aibo Division*. The reason for the change of the robot platform and the discontinuation of the *Aibo Division* from 2009 onwards was the end of production of *Aibo* in 2006. Most of the teams in the newly established league were already active in the *Aibo Division*. Therefore, it is not remarkable that as in the *Aibo* league there are also teams in the new league that are familiar with XABSL and use it for the behavior control of their *Nao* robots. Namely two teams that apply XABSL are *BreDoBrothers* [23, 91] and *NaoTeamHumboldt* [40].

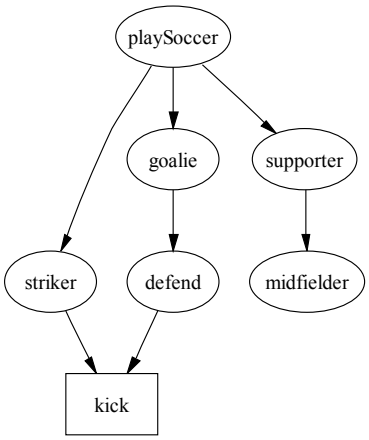
Even if most of these teams are directly connected to the *GermanTeam*, the large number of teams applying XABSL in different leagues, most of which are using completely different robot hardware, proves that XABSL can easily be ported to a variety of robotic



(a) A scene from an *ASCII Soccer* game.



(b) *SoccerBots* simulation with XABSL agents playing from right to left.



(c) Option graph of the *SoccerBots* demo.

Figure 6.19: XABSL demo applications.

platforms. In the next section it is shown that XABSL is also not at all restricted to the application domain of robot soccer.

6.3 Other Applications

In the next section it is shown that XABSL is not only suited for *RoboCup* soccer applications. Other applications include different platforms and different application domains.

6.3.1 Soccer Demos

Two small demo applications that are not directly related to *RoboCup* but are still located in the robot soccer domain have been implemented in order to support behavior developers who want to employ XABSL.

The first example application was made for the *ASCII Soccer* simulator [9]. In this very simple soccer simulation the field, two teams of four players each, and the ball are displayed on a text terminal (cf. Figure 6.19(a)).

The ASCII Soccer XABSL example implementation can be downloaded together with the complete source code and tools from the XABSL web site [69].

The second demo application is a behavior implementation in the Java based graphical simulation environment *TeamBots*, which is a successor of *ASCII Soccer*. It contains a simulation of a simplified robot soccer scenario called *SoccerBots* simulating dynamics and rules of the *RoboCup Small Size League* [10]. The XABSL behavior implementation also is an example for the usage of the Java XABSL engine (cf. Figure 6.19(b) and 6.19(c)).

In both simulation environments, the players are able to access a nearly complete world model and the action sets of the agents are very limited. The simplicity of these environments made it possible to develop competitive XABSL example agent teams with dynamic role assignments, supporter positioning, passing, and dribbling in a short time. These implementations also demonstrate that the XABSL architecture, language, the tools and the executing engine are not only suited for real robot soccer environments.

6.3.2 Heterogeneous Cooperation Demo

An application outside of the robot soccer domain has been realized successfully in a case study of cooperating, strongly heterogeneous, autonomous robots: a humanoid robot of the *Darmstadt Dribblers* and a *Pioneer 2DX* wheeled robot (cf. Fig. 6.20). The strongly cooperative task in this scenario is to jointly follow a ball over a long distance. The humanoid robot is able to track the ball with a directed camera, while the wheeled robot is able to transport the humanoid robot over long distances [51, 52].

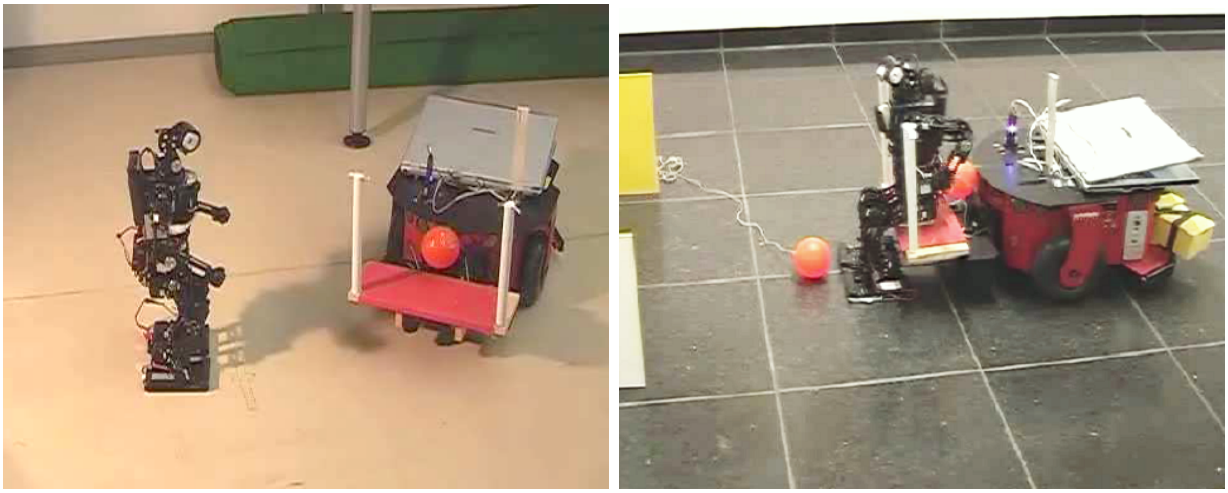
6.3.3 Darmstadt Rescue Robot Team

In *RoboCup* there are not only robot soccer competitions but there is also the *RoboCupRescue* competition. An application scenario possibly involving large teams of heterogeneous robots in a hostile environment can be found in disaster rescue. The *RoboCupRescue* project promotes research and development in this socially significant domain which involves many different aspects such as multi-agent team work coordination and physical robotic agents for search and rescue.

In 2009 the author's group will participate in *RoboCupRescue* with an autonomous four-wheeled robot [4]. One of the robots of the new team is shown in Figure 6.21. The team focuses on the autonomous operation of their vehicle. For high-level behavior decision making XABSL is being applied. Tasks solved with XABSL include exploration of an unknown area, navigation towards points of interest, and control of the viewing direction of the directed camera system.

6.3.4 Upcoming Applications

In the near future the list of real-world robot applications realized with behavior control based on hierarchical state machines will certainly grow further. For instance, in the author's group there is ongoing research on a number of different robotic platforms. As



(a) Humanoid robot approaches wheeled robot in order to board and get transported.

(b) Wheeled robot transports humanoid robot, while humanoid robot is tracking the ball and guides the wheeled robot. Humanoid robot is about to dismount from the wheeled robot in order to be able to manipulate the ball.

Figure 6.20: Example of strongly heterogeneous multi-robot cooperation using XABSL [52].

Figure 6.21: The robotic vehicle of the *Darmstadt Rescue Robot Team*.

Figure 6.22: Other robotic platforms under research at the author's group.

soon as complex autonomous tasks are being addressed using these platforms, it is planned to apply XABSL for high-level behavior control. Examples of robots that are under development in current research projects include a lightweight robot arm with a novel compliant design [54], an unmanned aerial vehicle, and an unmanned marine vehicle [25] (cf. Figure 6.22).

7 Conclusions and Outlook

In this thesis methodological extensions are presented which were applied to a behavior control architecture based on hierarchical finite state machines in Chapter 4. The resulting extended version of the *Extensible Agent Behavior Specification Language* (XABSL) was described in Chapter 5. The different applications examples given in Chapter 6 indicate that the design requirements stated in Chapter 3 have indeed been met.

The recent version of XABSL enables the convenient development of the behavior of autonomous agents even for very large and complex real-world robot applications. Techniques based on hierarchical state machines allow for efficient agent behavior dealing with uncertainty in highly dynamic environments. Composing state machine based options in hierarchies makes behaviors reusable in different contexts and thus enables behavior designers to develop scalable and highly complex behaviors.

The modular nature of XABSL supports the development of behaviors by a large team of programmers. New options can be easily added to existing ones without having negative side effects. With the debugging interfaces of the *XabslEngine* new options can be tested separately before they are used by higher-level options. Improved versions of existing options can be developed in parallel and are easy to compare with previous ones. A constantly growing library of well tuned low-level behaviors can be reused in different contexts for the creation of new options.

XABSL is becoming increasingly wide spread. Today, it is used by several teams in the *RoboCup Standard Platform League*, and it is also applied on other robots in the *RoboCup Middle Size*, *Small Size*, and *Humanoid League*. It helped the *GermanTeam* to become the 2004, 2005, and 2008 world champions in the *Standard Platform League*. Although this success was of course based on many other achievements as well, we believe that the ability of the team to develop and adopt very complex and efficient behaviors – even during the ongoing competition – played a key role in winning these titles.

Although XABSL was initially developed for robotic soccer, it is not a soccer programming language – there are no language elements of concepts that are specific to soccer applications. The language and the runtime system *XabslEngine* are application and platform independent and can be relatively easily employed in any agent system. An example of a successful application outside the domain of robot soccer can be found in the *Darmstadt Rescue Robot Team*.

Future work includes continued investigation of application domains outside of robot soccer, applications running on all kinds of robotic hardware, and large, complex, and possibly heterogeneous multi-robot cooperations. The combination of machine learning algorithms and behavior programmed in hierarchical state machines should be studied further: Behaviors which make intensive use of the layered learning paradigm, applying machine learning for different subtasks on different hierarchy levels, are promising directions for further research. Mixed integer optimization methods that do not only determine continuous parameter values but also include discrete values also might prove serviceable.

Formal analysis of hierarchical state machines can be applied in order to examine complex behaviors. For instance, through the application of model checking the correctness of certain properties of the behavior could be verified.

Bibliography

[1] Aldebaran. 'Aldebaran homepage.', 2009. http://www.aldebaran-robotics.com/eng/index.php.

[2] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. 'Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems.' In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. 1993.

[3] R. Alur and M. Yannakakis. 'Model Checking of Hierarchical State Machines.' In *In Symposium on the Foundations of Software Engineering*, pages 175–188. 1998.

[4] M. Andriluka, M. Friedmann, S. Kohlbrecher, J. Meyer, K. Petersen, C. Reinl, P. Schauss, P. Schnitzspan, A. Strobel, D. Thomas, and O. von Stryk. 'RoboCupRescue 2009 – Robot League Team – Darmstadt Rescue Robot Team (Germany).' In *RoboCup 2009: Robot Soccer World Cup XIII*, Lecture Notes in Artificial Intelligence. Springer, 2009. To appear.

[5] T. Arai and F. Stolzenburg. 'Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing.' In *In Proceedings of the 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems*, pages 11–18. ACM Press, 2002.

[6] R. C. Arkin. 'Motor Schema-Based Mobile Robot Navigation.' *The International Journal of Robotics Research*, 8(4), 1989.

[7] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[8] B. Bakker and J. Schmidhuber. 'Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization.' In *Proceedings of IAS-8*, pages 438–445. 2004.

[9] T. Balch. 'The Ascii Robot Soccer Homepage.', 1995. http://www-2.cs.cmu.edu/~trb/soccer/.

[10] T. Balch. 'SoccerBots web site.', 2000. http://www.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots/index.html.

[11] D. Becker, J. Brose, D. Göhring, M. Jüngel, M. Risler, and T. Röfer. 'GermanTeam 2008 - The German National RoboCup Team.' *Technical report*, DFKI Bremen, TU Darmstadt, HU Berlin, 2008.

[12] D. Becker and M. Risler. 'Mutual Localization in a Team of Autonomous Robots Using Acoustic Robot Detection.' In *RoboCup 2008: Robot Soccer World Cup XII*, Lecture Notes in Artificial Intelligence. Springer, 2009. To appear.

[13] M. Beetz and D. McDermott. 'Executing Structured Reactive Plans.' In *Proceedings of the AAAI Fall Symposium: Issues in Plan Execution*. AAAI, 1996.

[14] R. Berger. 'Die Doppelpass-Architektur – Verhaltenssteuerung autonomer Agenten in dynamischen Umgebungen.' Diploma thesis. Humboldt-Universität zu Berlin, 2006.

[15] O. Birbach, A. Burchardt, C. Elfers, T. Laue, F. Penquitt, T. Schindler, and K. Stoye. 'B-Smart. Team Description for RoboCup 2006.' In *RoboCup 2006: Robot Soccer World Cup X*. 2007.

[16] M. E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999.

[17] R. A. Brooks. 'A Robust Layered Control System for a Mobile Robot.' *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[18] R. A. Brooks. 'The Behavior Language; User's Guide.' *Technical Report AIM-1127, MIT Artificial Intelligence Lab*, 1990.

[19] R. A. Brooks. 'Intelligence without Representation.' *Artificial Intelligence*, 47:139–159, 1991.

[20] H.-D. Burkhard. 'Programming Bounded Rationality.' In *Proceedings of the International Workshop on Monitoring, Security, and Rescue Techniques in Multiagent Systems (MSRAS 2004)*, pages 347–362. Springer, 2005.

[21] H.-D. Burkhard, J. Bach, R. Berger, B. Brunswiek, and M. Gollin. 'Mental Models for Robot Control.' In *Advances in Plan-Based Control of Robotic Agents*, volume 2466 of *Lecture Notes in Artificial Intelligence*, pages 71–88. Springer, 2002.

[22] P. Currier, J. G. Hurdus, A. Wicks, and C. Reinholtz. 'Team Victor Tango's Odin: Autonomous Driving Using NI LabVIEW in the DARPA Urban Challenge.' case study at National Instruments web site, 2009. Available online: `http://sine.ni.com/cs/app/doc/p/id/cs-11323`.

[23] S. Czarnetzki, D. Hauschildt, S. Kerner, and O. Urbann. 'BreDoBrothers Team Report 2008.', 2008. Only available online: `http://www.tzi.de/spl/pub/Website/Teams2008/BreDoBrothers.pdf`.

[24] S. Czarnetzki, M. Hebbel, and W. Nisticò. 'DoH!Bots Team Description for RoboCup 2007.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[25] C. Ecke. 'Sensorik, Regelung und Realisierung eines unbemannten schwimmenden Fahrzeuges.' Diploma thesis. Technische Universität Darmstadt, 2008.

[26] J. Ellson, E. Gansner, and Y. Hu. 'GraphViz Web Site.', 2008. `http://www.graphviz.org`.

[27] P. Fidelman and P. Stone. 'Learning Ball Acquisition on a Physical Robot.' In *In 2004 International Symposium on Robotics and Automation (ISRA)*. 2004.

[28] P. Fidelman and P. Stone. 'The Chin Pinch: A Case Study in Skill Learning on a Legged Robot.' In G. Lakemeyer, E. Sklar, D. Sorenti, and T. Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup X*, pages 59–71. Springer Verlag, 2007.

[29] M. Friedmann, J. Kiener, R. Kratz, T. Ludwig, S. Petters, M. Stelzer, O. von Stryk, and D. Thomas. 'Darmstadt Dribblers 2005: Humanoid Robot (Team Description Paper).' *Technical report*, Technische Universität Darmstadt, 2005.

[30] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. 'Modular Software Architecture for Teams of Cooperating, Heterogeneous Robots.' In *Proc. IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 613–618. Kunming, China, December 17-20 2006.

[31] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. 'Reusable Architecture and Tools for Teams of Lightweight Heterogeneous Robots.' In *Proc. 1st IFAC Workshop on Multivehicle Systems*, pages 51–56. Salvador, Brazil, October 2-3 2006.

[32] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. 'Darmstadt Dribblers: Team Description for Humanoid KidSize League of RoboCup 2007.' *Technical report*, Technisch Universität Darmstadt, 2007.

[33] M. Friedmann, K. Petersen, S. Petters, K. Radkhah, D. Thomas, and O. von Stryk. 'Darmstadt Dribblers: Team Description for Humanoid KidSize League of RoboCup 2008.' *Technical report*, Technische Universität Darmstadt, 2008.

[34] M. Fujita and H. Kitano. 'Development of an Autonomous Quadruped Robot for Robot Entertainment.' *Auton. Robots*, 5(1):7–18, 1998.

[35] U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg. 'Hybrid Multiagent Systems with Timed Synchronization – Specification and Model Checking.' In *Proceedings of 5th International Workshop on Programming Multi-Agent Systems*, pages 170–185. Honolulu, Hawaii, 2007.

[36] E. R. Gansner and S. C. North. 'An Open Graph Visualization System and its Applications to Software Engineering.' *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[37] E. Gat. 'Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots.' In *Proceedings AAAI-92*, pages 809–815. MIT Press, 1992.

[38] T. Gindele, D. Jagszent, B. Pitzer, and R. Dillmann. 'Design of the planner of Team AnnieWAY's autonomous vehicle used in the DARPA Urban Challenge 2007.' In *Intelligent Vehicles Symposium*. IEEE, Eindhoven, The Netherlands, 2008.

[39] D. Göhring. 'Cooperative Object Localization Using Line-Based Percept Communication.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[40] D. Göhring. 'NaoTeamHumboldt Web Site.', 2008. `http://www.naoteamhumboldt.de`.

[41] D. Göhring, H. Mellmann, and H.-D. Burkhard. 'Constraint Based Belief Modeling.' In *RoboCup 2008: Robot Soccer World Cup XII*, Lecture Notes in Artificial Intelligence. Springer, 2009. To appear.

[42] G. A. Gray and T. G. Kolda. 'Algorithm 856: APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization.' *ACM Transactions on Mathematical Software*, 32(3):485–507, 2006.

[43] D. Hähnel, W. Burgard, and G. Lakemeyer. 'GOLEX - Bridging the Gap between Logic (GOLOG) and a Real Robot.' In *KI '98: Proceedings of the 22nd Annual German Conference on Artificial Intelligence*, pages 165–176. Springer-Verlag, 1998.

[44] D. Harel. 'Statecharts: A Visual Formalism for Complex Systems.' *Science of Computer Programming*, 8(3):231–274, June 1987.

[45] M. Hebbel, W. Nistico, T. Kerkhof, M. Meyer, C. Neng, M. Schallaböck, M. Wachter, J. Wege, and C. Zarges. 'Microsoft Hellhounds 2006.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[46] M. Hild, M. Jüngel, and M. Spranger. 'Humanoid Team Humboldt Team Description 2006.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[47] J. G. Hurdus. 'A Portable Approach to High-Level Behavioral Programming for Complex Autonomous Robot Applications.' Master's thesis. Virginia Polytechnic Institute and State University, 2008.

[48] M. Jüngel, H. Mellmann, and M. Spranger. 'Improving Vision-Based Distance Measurements Using Reference Objects.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[49] M. Jüngel and M. Risler. 'Self-Localization Using Odometry and Horizontal Bearings to Landmarks.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[50] S. Kammel, J. Ziegler, B. Pitzer, M. Werling, T. Gindele, D. Jagszent, J. Schröder, M. Thuy, M. Goebl, F. von Hundelshausen, O. Pink, C. Frese, and C. Stiller. 'Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge.' *J. Field Robot.*, 25(9):615–639, 2008.

[51] J. Kiener. *Heterogene Teams kooperierender autonomer Roboter (Heterogeneous Teams of Cooperating Robots).* Number 1128 in Reihe 8: Meß-, Steuerungs- und Regelungstechnik. VDI Verlag, 2007.

[52] J. Kiener and O. von Stryk. 'Cooperation of Heterogeneous, Autonomous Robots: A Case Study of Humanoid and Wheeled Robots.' In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 959–964. San Diego, USA, 2007.

[53] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. 'RoboCup: The Robot World Cup Initiative.' In *Proceedings of Agents'97*, pages 340–347. ACM Press, 1997.

[54] S. Klug, T. Lens, O. von Stryk, B. Möhl, and A. Karguth. 'Biologically Inspired Robot Manipulator for New Applications in Automation Engineering.' In *Proceedings of Robotik 2008*, number 2012 in VDI-Berichte. VDI Wissensforum GmbH, Munich, Germany, June 11-12 2008.

[55] H. Kobayashi, T. Osaki, E. Williams, A. Ishino, and A. Shinohara. 'Autonomous Learning of Ball Trapping in the Four-Legged Robot League.' pages 86–97, 2007.

[56] B. Koch. 'Hamburg Dog Bots - Team Description Paper 2006.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[57] T. G. Kolda. 'Revisiting Asynchronous Parallel Pattern Search for Nonlinear Optimization.' *SIAM Journal on Optimization*, 16(2):563–586, December 2005.

[58] K. Konolige. 'COLBERT: A Language for Reactive Control in Sapphira.' In *KI-97: Advances in Artificial Intelligence*, number 1303 in LNAI, pages 31–52. Springer, 1997.

[59] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti. 'The Saphira Architecture: A Design for Autonomy.' *Journal of Experimental and Theoretical Artificial Intelligence*, 9:215–235, 1997.

[60] R. Lafrenz, O. Zweigle, U.-P. Käppeler, H. Rajaie, A. Tamke, T. Rühr, M. Oubbati, M. Schanz, F. Schreiber, and P. Levi. 'Major Scientific Achievements 2006 - CoPS Stuttgart Registering for World Championships in Bremen.' In *RoboCup International Symposium 2006*, pages 1–9. Bremen: Università degli Studi di Milano, 2006.

[61] T. Laue and T. Röfer. 'A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields.' In *In 8th International. Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence, Lecture Notes in Computer Science*, pages 122–133. Springer, 2005.

[62] T. Laue and T. Röfer. 'Getting Upright: Migrating Concepts and Software from Four-Legged to Humanoid Soccer Robots.' In E. Pagello, C. Zhou, and E. Menegatti, editors, *Proceedings of the Workshop on Humanoid Soccer Robots in conjunction with the 2006 IEEE International Conference on Humanoid Robots*. 2006.

[63] T. Laue, K. Spiess, and T. Röfer. 'SimRobot - A General Physical Robot Simulator and Its Application in RoboCup.' In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, number 4020 in Lecture Notes in Artificial Intelligence, pages 173–183. Springer, 2006.

[64] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. 'GOLOG: A Logic Programming Language for Dynamic Domains.' *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[65] M. Lötzsch. 'XABSL - A Behavior Engineering System for Autonomous Agents.' Diploma thesis. Humboldt-Universität zu Berlin, 2004. Available online: `http://www.martin-loetzsch.de/papers/diploma-thesis.pdf`.

[66] M. Lötzsch. 'DotML Web Site.', 2006. `http://www.martin-loetzsch.de/DOTML/`.

[67] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. 'Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL.' In *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *LNAI*, pages 114–124. Springer, 2004.

[68] M. Lötzsch, M. Risler, and M. Jüngel. 'XABSL - A Pragmatic Approach to Behavior Engineering.' In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129. Beijing, China, 2006.

[69] M. Lötzsch, M. Risler, and M. Jüngel. 'XABSL Web Site.', 2008. `http://www.xabsl.de`.

[70] D. MacKenzie, R. Arkin, and J. Cameron. 'Multiagent Mission Specification and Execution.' *Autonomous Robots*, 4(1):29–52, 1997.

[71] P. Maes. 'Situated Agents Can Have Goals.' In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 49–70. MIT Press, 1990.

[72] E. Martinson, A. Stoytchev, and R. Arkin. 'Robot Behavioral Selection using Q-Learning.' In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*. 2002.

[73] J. Mccarthy and P. J. Hayes. 'Some Philosophical Problems from the Standpoint of Artificial Intelligence.' In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[74] D. McDermott. 'A Reactive Plan Language.' *Technical report*, 1993.

[75] D. McDermott. 'PDDL: The Planning Domain Definition Language.' *Technical report*, 1998.

[76] M. Minsky. *The Society of Mind*. Simon and Schuster, 1986.

[77] J. Murray. 'Specifying Agents with UML Statecharts and StatEdit.' In A. Bonarini, B. Browning, D. Polani, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 145–156. Springer, 2004.

[78] J. Murray and F. Stolzenburg. 'Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification.' In C. Bento, A. Cardoso, and G. Dias, editors, *Proceedings of 12th Portuguese Conference on Artificial Intelligence*, pages 236–241. Institute of Electrical and Electronics Engineers (IEEE), Inc., 2005. ISBN 0-7803-9365-1.

[79] Object Management Group. 'Unified Modeling Language (UML), version 2.2.', 2009. `http://www.omg.org/technology/documents/formal/uml.htm`.

[80] S. Oomes, P. Jonker, M. Poel, A. Visser, and M. Wiering. 'The Dutch AIBO Team 2004.' In *RoboCup 2004: Robot Soccer World Cup VIII*, LNAI. Springer, 2005.

[81] S. Petters and D. Thomas. 'RoboFrame - Softwareframework für mobile autonome Robotersysteme.' Diploma thesis. Technische Universität Darmstadt, 2005.

[82] S. Petters, D. Thomas, and O. von Stryk. 'RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots.' In *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*. San Diego, CA, USA, Oct. 29 2007.

[83] M. Risler and O. von Stryk. 'Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL.' In *AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*. Estoril, Portugal, 2008.

[84] T. Röfer. 'Region-Based Segmentation With Ambiguous Color Classes And 2-D Motion Compensation.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[85] T. Röfer and et al. 'GermanTeam 2002.' In *RoboCup 2002: Robot Soccer World Cup VI*. RoboCup Federation, 2003.

[86] T. Röfer and et al. 'GermanTeam Web Site.', 2008. `http://www.germanteam.org`.

[87] T. Röfer, J. Brose, E. Carls, J. Carstens, D. Göhring, M. Jüngel, T. Laue, T. Oberlies, S. Oesau, M. Risler, M. Spranger, C. Werner, and J. Zimmer. 'GermanTeam 2006 - The German National RoboCup Team.' *Technical report*, DFKI Bremen, Universität Bremen, TU Darmstadt, HU Berlin, 2006.

[88] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. 'GermanTeam 2007 - The German National RoboCup Team.' *Technical report*, DFKI Bremen, TU Darmstadt, HU Berlin, Universität Bremen, 2007.

[89] T. Röfer, C. Budelmann, M. Fritsche, T. Laue, J. Müller, C. Niehaus, and F. Penquitt. 'B-Human Team Description for RoboCup 2007.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[90] T. Röfer, M. Fritsche, M. Hebbel, T. Kindler, T. Laue, C. Niehaus, W. Nistico, and P. Schober. 'BreDoBrothers Team Description for RoboCup 2006.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[91] T. Röfer, T. Laue, E. Damrose, K. Gillmann, T. de Haas, A. Härtl, A. Rieskamp, A. Schreck, and J. Worch. 'B-Human Team Report and Code Release 2008.', 2008. Only available online: `http://www.b-human.de/download.php?file=coderelease08_doc`.

[92] T. Röfer, T. Laue, M. Weber, H.-D. Burkhard, M. Jüngel, D. Göhring, J. Hoffmann, B. Altmeyer, T. Krause, M. Spranger, O. von Stryk, R. Brunn, M. Dassler, M. Kunz, T. Oberlies, M. Risler, U. Schwiegelshohn, M. Hebbel, W. Nistico, S. Czarnetzki, T. Kerkhof, M. Meyer, C. Rohde, B. Schmitz, M. Wachter, T. Wegner, and C. Zarges. 'GermanTeam 2005.' *Technical report*, Humboldt Universität zu Berlin, Universität Bremen, Techniche Universität Darmstadt, Universität Dortmund, 2005.

[93] F. Ruh. 'A Translator for Cooperative Strategies of Mobile Agents for Four-Legged Robots.' Master's thesis. Hochschule Harz, 2007.

[94] G. A. Rummery and M. Niranjan. 'On-line Q-Learning using Connectionist Systems.' *Technical report*, 1994.

[95] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, 1995.

[96] S. P. Singh, R. S. Sutton, and P. Kaelbling. 'Reinforcement Learning with Replacing Eligibility Traces.' In *Machine Learning*, pages 123–158. 1996.

[97] P. Stone and M. Veloso. 'Layered Learning.' In *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain, May 31 - June 2, 2000, Proceedings*, volume 1810, pages 369–381. Springer, Berlin, 2000.

[98] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1999.

[99] J. J. Tapia, C. Reyes, J. R. Uresti, A. Sobrino, E. Cruz, E. Millan, G. Villareal, J. Gordon, L. Toledo, A. Morales, M. Silva, M. Maqueo, M. Coronel, M. Arias, V. Ferman, W. Cabrera, and J. Vano. 'TecRams: Research Work Done in Robotics at Tecnologico de Monterrey – Team Description Paper.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[100] S. Thrun, M. Bennewitz, W. Burgard, A. Cremers, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. 'MINERVA: A second generation mobile tour-guide robot.' In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA'99)*. 1999.

[101] A. Visser, P. van Rossum, J. Westra, J. Sturm, D. van Soest, and M. de Greef. 'Dutch AIBO Team at RoboCup 2006.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[102] Q. Wang, C. Rong, G. Xie, and L. Wang. 'Team Description of sharPKUngfu 2006.' In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2008.

[103] C. J. C. H. Watkins. *Learning from Delayed Rewards.* Ph.D. thesis, King's College, Cambridge, UK, 1989.

[104] S. Whiteson and P. Stone. 'Concurrent Layered Learning.' In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 193–200. ACM Press, New York, NY, July 2003.

[105] M. Wooldridge. *Reasoning about Rational Agents.* The MIT Press, June 2000.

[106] K. Xu, X. Zhang, X. Liu, F. Liu, H. Zhang, B. Ren, H. He, F. Wang, and X. Chen. 'Wrighteagle 2006 Team Description.' In *RoboCup 2006: Robot Soccer World Cup X*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[107] V. A. Ziparo and L. Iocchi. 'Petri Net Plans.' In *Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA'06) @ Petri Nets 2006 and ACSD 2006*, pages 267–289. Turku, Finland, 2006.

[108] O. Zweigle, R. Lafrenz, T. Buchheim, U.-P. Käppeler, H. Rajaie, F. Schreiber, and P.Levi. 'Cooperative Agent Behavior Based on Special Interaction Nets.' In *IAS*, pages 651–659. IOS Press, 2006.

Lebenslauf

07/1990–06/1997	Kooperative Gesamtschule Leeste, Weyhe
06/1997	Allgemeine Hochschulreife
10/1998–04/2004	Studium der Informatik an der Technischen Universität Darmstadt Nebenfach: Physik
04/2004	Diplom in Informatik
05/2004–06/2005	Hilfskraft mit Abschluss, Fachbereich Informatik, Technische Universität Darmstadt
seit 07/2005	Wissenschaftlicher Mitarbeiter, Fachbereich Informatik, Technische Universität Darmstadt