Humboldt-Universität zu Berlin

Department of Computer Science

Research Group Artificial Intelligence

Martin Lötzsch

XABSL - A Behavior Engineering System for Autonomous Agents

Diploma Thesis

Advisors: Prof. Hans-Dieter Burkhard

Dr. Thomas Röfer

October 2004

Erklärung

Hiermit erkläre ich, die vorliegende Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst zu haben.

Ich bin damit einverstanden, dass ein Exemplar dieser Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt wird.

Berlin, Oktober 2004

Martin Lötzsch

Abstract. In the area of agent systems, as throughout in computer science, formal methods are applied to specify complex systems, to ensure certain properties of a system, or to generally simplify the development of solutions. In traditional symbolic artificial intelligence, logic and planning theories are usually used for modelling autonomous agents. But these approaches turned out to be not applicable for agent systems in highly dynamic environments. Therefore, mainly in intelligent robotics, theories or formalisms are often not used when programming agents to perform certain tasks. In this thesis, the *Extensible Agent Behavior Specification Language* (*XABSL*) is introduced as a pragmatic approach to agent engineering. It does not follow any agent theory but in return provides a powerful set of tools for the convenient and rapid programming of agent behavior. The system was succesfully applied by the *GermanTeam* in the RoboCup Sony Four Legged League, resulting in the win of the RoboCup world championship 2004.

Contents

1 Introduction

Multi-agent systems in complex and dynamic environments are a more and more important research subject both in intelligent robotics and artificial intelligence. In traditional robotics, impressing behaviors have been realized with simple sensor-actuator loops. Although Brooks [13] showed how to combine these control programs to achieve more complex behaviors, it is challenging to scale up such systems.

Many agent architectures from classical AI were developed for simplified and artificial environments. Therefore problems arise when they are confronted with more natural environments. Noisy sensor readings, unpredictable dynamics, and the uncertainty of actions ask for new sophisticated approaches.

1.1 Goals of this Work

The major goals of this work are the design of a behavior control architecture for autonomous agents in highly dynamic environments as well as the development of a high-level language for describing agent behavior following such an architecture. The RoboCup domain [36], a common problem for multi-agent systems, provides a test bed for that work.

The architecture has to support complex long-term and deliberative decision processes as well as short-term reactive behaviors. Moreover, it is important that the architecture is able to deal with uncertain environments where actions can fail. It is necessary that the system is modular to ensure the reusability of behaviors in different contexts and the extensibility of implementations. The language has to simplify and speed up the process of agent behavior specification. It should be scalable and easy to understand. Finally, it should help behavior designers to keep an overview over large behavior implementations.

1.2 Scope

Many agent theories or agent architectures deal with the description of whole agent systems, including their perception and action capabilities and possibilities to reason how to achieve goals. For example, in the

BDI architecture [59, 60] the *beliefs*, *desires*, and *intentions* of an agent are modeled. The *beliefs* represent information items of the environment's state and are updated after each sensing action. The *desires* of an agent represent the objectives (or goals) and what priorities or tradeoffs are associated with these objectives. Reasoning about the *desires*, the *selection function* determines the system's *intentions*. The *actions* of the system are generated based on the *intentions*.

Some AI researchers approach the problem of generating *intentions* by *desires* with modal logic [18, 70]. They try to formalize all interactions between the *beliefs*, *desires*, and *intentions*, all static and dynamical constraints of the system, and the impact of the actions on the environment. If such a formalization exists, intentions can be achieved by applying decision theory. Generating appropriate actions can be seen as a kind of classical problem solving. However, for agents in complex, highly dynamic, and unpredictable environments it is a difficult task to cope with the dynamics of the system by means of logic. Logic based planning algorithms are therefore not in the scope of this work. Gat [27] remarks: "Elevator doors and oncoming trucks wait for no theorem prover."

This work deals with the selection of actions from beliefs. Deliberative and reactive decision making is based on hierarchical plan structures which are pre-defined by behavior designers. It focuses neither on how the *beliefs* of an agent are structured or how they are created nor on how the actions of the agent are performed. *Beliefs* and actuator control programs are assumed to exist already in the agent system.

Wooldridge [73, 72] divides the field of intelligent agents into the areas of agent theories, agent architectures, and agent languages. Agent theories are not in the scope of this work and as holistic agents are not investigated but only the selection of actions, the terms *behavior control architectures* and *behavior control languages* are used instead of agent architectures and agent languages.

Although the work is related to multi agent systems, it does not deal explicitly with the modeling of agent communication and negotiation issues (cf. e.g. [30, 69]). Messages that are exchanged between agents can be seen as additional input and output of an action selection system. It will be shown how agent teams modeled with the proposed architecture will perform cooperative tasks using communication.

Many approaches deal with either deliberative or reactive path planning. But behaviors which perform obstacle avoidance or other navigation tasks using such algorithms can be seen as basic skills of an agent system and will therefore not be dealt with in this work, too.

1.3 Outline

Chapter 2 gives an overview of state-of-the-art agent and behavior control architectures and languages. Chapter 3 is the central part of this work. It describes a behavior control architecture based on hierarchical finite state machines[1] for action selection. The *Extensible Agent Behavior Specification Language* (*XABSL*) is introduced as an XML based agent language formalizing that architecture. The language itself, the tools that were developed in conjunction with the language, and the runtime system *XabslEngine* are described. Although the *XABSL* system is fully explained in this thesis, there is an even more detailed language reference and the API documentation of the *XabslEngine* class library on the *XABSL* web site [45].

Chapter 4 shows how the *XABSL* system was applied in the RoboCup robot soccer domain. The *GermanTeam*, a national team consisting of researchers from four German universities, uses *XABSL* for its participation in the RoboCup Sony Four Legged League [62]. *XABSL* helped the team to win the 2004 RoboCup world championships. In addition, an *XABSL* example implementation was done for the *ASCII-Soccer* [10] soccer simulation to show that the whole system is independent from the platform of the *GermanTeam*.

Finally, chapter 5 lists the results of *XABSL* based teams at national and international RoboCup competitions and suggests a few ideas for further improvements.

1.4 Own Contributions

The author developed the *XABSL* language definition, the tools, and the *XabslEngine* class library. Furthermore, the author was the leader of the behavior control group of the *GermanTeam* and coordinated its behavior control related attempts.

The behavior architecture was developed in collaboration with Hans-Dieter Burkhard, Matthias Jüngel, Joscha Bach, Ralf Berger, and Michael Gollin. Matthias Jüngel helped to implement some of the tools. Uwe Düffert and Thomas Röfer provided technical advice and bug fixes. Michael Spranger developed a profiling tool on top of the *XABSL* system. Max Risler and Matthias Jüngel made suggestions how to improve the language.

Finally, *XABSL* could not have been employed in the Sony Four Legged League had not numerous members of the *GermanTeam* implemented and tuned many behaviors.

[1] Instead of *finite state machine*, the term *finite state automaton* (*FSA*) is sometimes used in the literature. Nevertheless, for consistency with recent own publications and documentations, the term *finite state machine* will be consequently employed in this thesis.

Parts of this work (especially parts of chapter 3 and 4) have already been published by the author in [46, 45, 15, 20, 62].

2 Architectures and Languages

The field of behavior control architectures and behavior control languages is very wide. This chapter deals with those who are in a closer relation to this work. It will be shown that the distinction between architectures and languages is at times arbitrary as languages are always based on architectures and some architectures contain precise formalization.

2.1 Behavior Control Architectures

“Agent architectures can be thought of as software engineering models of agents; researchers in this area are primarily concerned with the problem of designing software or systems that will satisfy the properties specified by agent theorists.” [72]

Nearly all behavior control architectures are labeled either *reactive* or *deliberative* or both. But there are as many different usages of these terms as there are different architectures. The weakest notion of *deliberative* requires that the environment is represented in a persistent state, the world model. Accordingly, all behaviors that react directly on the sensory inputs are labeled reactive.

Deliberative in a stronger sense means that persistent states of own intentions are used. Decisions are made not only dependent on a world model but also based on past decisions, which allows to continue started plans.

The strongest notion of deliberative means that an agent is able to develop abstract plans by reasoning about the state of the environment, own desires, own action capabilities and their impact on the environment. This notion is usually related to the term *planning*.

Hybrid architectures are employed when both fast adaptions to changes in the environment as well as deliberative behaviors are needed. In these architectures, a strict separation into a deliberative and a reactive component is done. However, it is problematic that there is no general rule that assigns a behavior to one of these components. Usually, this separation is done dependent on the time consumption of the decision making methods. All decisions that are more time consuming than allowed in the reactive component are

made by the deliberative component. Architectures in which such a separation is not done but which are nevertheless able to model both very long-term and short-term behaviors, are hard to classify but are often labeled reactive.

Besides the distinction in reactive and deliberative, Russel and Norvig [66] propose a different categorization of agent architectures: *Simple reflex agents* respond immediately to percepts, *goal based agents* act so that they will achieve their goals, and *utility based agents* try to maximize their own "happiness".

2.1.1 Behavior-Based Architectures

In reactive *behavior-based* architectures [7], behavior control programs are decomposed into a set of distinct low-level basic skills (basic behaviors) and selection mechanisms that combine these abilities into complex behaviors.

Each basic behavior is designated for performing a certain specific task. Mostly they are reactive, which means they do not have any persistent states but react directly to changes in the environment in close sensor-actuator loops. There are at least three different methods for combining or composing these behaviors.

Continuous Combination of Behaviors. Primitive behaviors can be combined continuously. When following this approach, all behaviors contribute to the entire agent behavior. For each of them an utility weight is estimated and then the output values of all behaviors are scaled by their corresponding weights and simply summed up. For instance, in a mobile robot several navigation behaviors (e.g. "move-to", "avoid-obstacles", "avoid-wall", etc.) could all provide their desired movement vectors. By scaling them with their utility weights and summing them up, an overall behavior that reaches a goal, avoids obstacles on the way there, and keeps distance from walls could emerge.

The *AuRA* architecture [6] gives an example how to to apply this method. However, complex systems with unsuperimposable actions can not be modeled with this.

Competitive Approaches. Furthermore, the basic behaviors can compete for the control of the agent. Maes [49] developed an architecture where the primitive behaviors are called *competence modules*. Each of these modules has a set of pre- and post-conditions and has to provide an *activation level*, an utility measure for the module in a particular situation. The higher the activation level of a module, the more likely this module will influence the behavior of the agent. The modules are connected by successor, predecessor, and conflicter links in a *spreading activation network*. Active modules inhibit other modules connected by conflicter links and activate neighbored predecessor and successor modules. The major difficulty for

applying this architecture is to find appropriate activation functions for each module. It is very hard to tune these functions so that in all situations the most applicable module is activated most.

A similar and even simpler architecture is the *subsumption architecture* by Brooks [13]. In this architecture, the *behaviors* are layered in a hierarchy. Primitive behaviors (such as collision avoidance) reside on lower layers whereas more high-level behaviors are placed on top of them. Lower layers can inhibit higher layers to gain control over the behavior of the system.

Finite State Machines. Finally, basic behaviors can be composed with *state based* techniques [39]. In these architectures, only one of the basic behaviors is active and executed at a time. Behavior selection is done by using *finite state machines*. The behaviors correspond to states and are selected by transitions between the states. The transition functions are dependent on the active state, events, changes in time, and changes in the environment. Arkin [5] uses the term *temporal sequencing* for the method as the behaviors are performed in sequential order.

Finite state machines exhibit two important advantages: First, it is possible to define a hysteresis between two states. For example, if there is a transition from state s_1 to state s_2 when a certain variable exceeds a threshold t, there could be a transition from s_2 to s_1 when the variable falls below $t - \epsilon$. This helps to stabilize decisions based on noisy sensor readings. Second, as decisions are made different for each state, only useful successor behaviors are selected.

2.1.2 Hierarchical and Layered Architectures

Many architectures are called *hierarchical* or *layered*. Usually this means that there are different levels of abstraction. Lower layers react directly on changes in the sensor readings. The higher the layer, the more long-term decisions are made and more abstract representations of the environment are used. Often, a separation into a deliberative and a reactive layer is done. Sometimes, these layers do not work synchronously, which means that high-level components are not executed as frequent as low-level ones.

Hierarchical can also mean, that many behaviors are ordered in a hierarchy. This can be seen from two perspectives: Either, top-level goals or plans can be recursively decomposed into sub goals or sub plans or, more high-level and complex behaviors can be composed from more low-level and simpler ones (modularity principle). This allows to reuse behaviors in different more high-level contexts. Systems can be easier developed as behaviors can be tested separately before they are composed to more complex ones. In addition, such modularity reduces the complexity of planning. It would be difficult to cope with if all decisions

would be made by a holistic system. Behavior hierarchies allow to distribute different decisions into different modules.

When using finite state machines for decision making, the number of transitions usually exponentially increases with the number of states. Ordering finite state machines in a hierarchy can reduce the number of necessary transitions.

2.2 Behavior Control Languages

Specific behavior description languages prove to be suitable replacements to native programming languages such as C++ when the number and complexity of behavior patterns of an agent increases. They allow for convenient and fast behavior design and implementations are often easier to scale-up. Visualizations and other helper tools support the development process. Additionally, some languages provide mechanisms to prove or guarantee certain agent properties. “Without adequate techniques to support the design process, such systems will not be sufficiently reliable, maintainable or extensible, will be difficult to comprehend, and their elements will not be re-usable.” [35]

There are innumerable languages that were developed for certain architectures and platforms. This section only lists a few noted languages. For an overview and introductions, refer to [47, 53, 31, 42].

2.2.1 The Behavior Language by Brooks

One of the oldest behavior specification languages, the *Behavior Language* [14], was developed by Brooks to specify agents following an improved version of the subsumption architecture [13]. It is a subset of Lisp and has a comparatively large expressivity. The behaviors are implemented in asynchronous processes which communicate via message passing. Each process contains a set of “real-time rules” that modify certain variables, send messages, or influence other behaviors. Initially, a process is in a wait state. As soon as the condition of a rule becomes true, the statements inside a rule are evaluated sequentially and the process returns into the wait state. If the rule recursively contains other rules, the control remains inside this rule. All rules are assumed to run in parallel and asynchronously. But there is also a possibility to declare exclusive rules. When the condition of such a rule becomes true, no other rules are evaluated.

Such a rule set can be compiled into a finite state machine, which is then either compiled into the assembler code of an embedded system or into Common Lisp code for simulation purposes. The compiler also organizes the serialization of the concurrent processes as real concurrency is often not possible on embedded

systems.

The language convinces by its complexity and well-chosen design. The rich set of usable mechanisms makes it a good choice for developing systems following the subsumption architectures.

2.2.2 COLBERT

Much simpler is the *COLBERT* language [37] which was developed by Konolige for reactive control in the *Saphira* [38] framework. *COLBERT* is a subset of ANSI C with a few extensions for robot control. An interpreter executes the language directly, so that programs can be modified during execution. A debugging tool allows for monitoring the current state activations. Source code in this language looks similar to usual C programs, for instance there are $while$ and if statements. Function calls in control blocks, that perform actions, correspond to states of a finite state machine. The control remains in the state until the behavior is finished. In addition, external conditions such as timeouts or other events can be specified to influence the control.

The language follows a simple and straightforward approach. However, because of its simplicity it could be difficult to apply it in more complex systems.

2.2.3 GOLOG

GOLOG by Lakemeyer and Levesque [43] is the most widely known logic based behavior control language. It is based on the situation calculus and has a syntax similar to Prolog, with extensions for procedural constructs. The actions of the system are generated by theorem proving as in a Prolog interpreter. Although Dylla et al. [22] showed how to use GOLOG on real robots, it is very hard to apply this language to systems in dynamic environments. The first problem is that the real world has to be translated into accurate symbolic descriptions based on logical terms, which can be difficult for uncertain environments. Second, the impact of actions on the environment and thereby on the world state of an agent has to be also represented with logic, which is even more difficult. Finally, speed is a problem when using Prolog interpreters in real-time systems.

2.2.4 UML Statecharts for Multiagent Specification

Obst and Stolzenburg et al. employ UML statecharts for multi-agent specification [58, 4, 54]. Hierarchies of state machines are used for action selection. Transition between states are equipped with simple variables and predicates connected by simple Boolean expressions. Although the authors claim that the UML

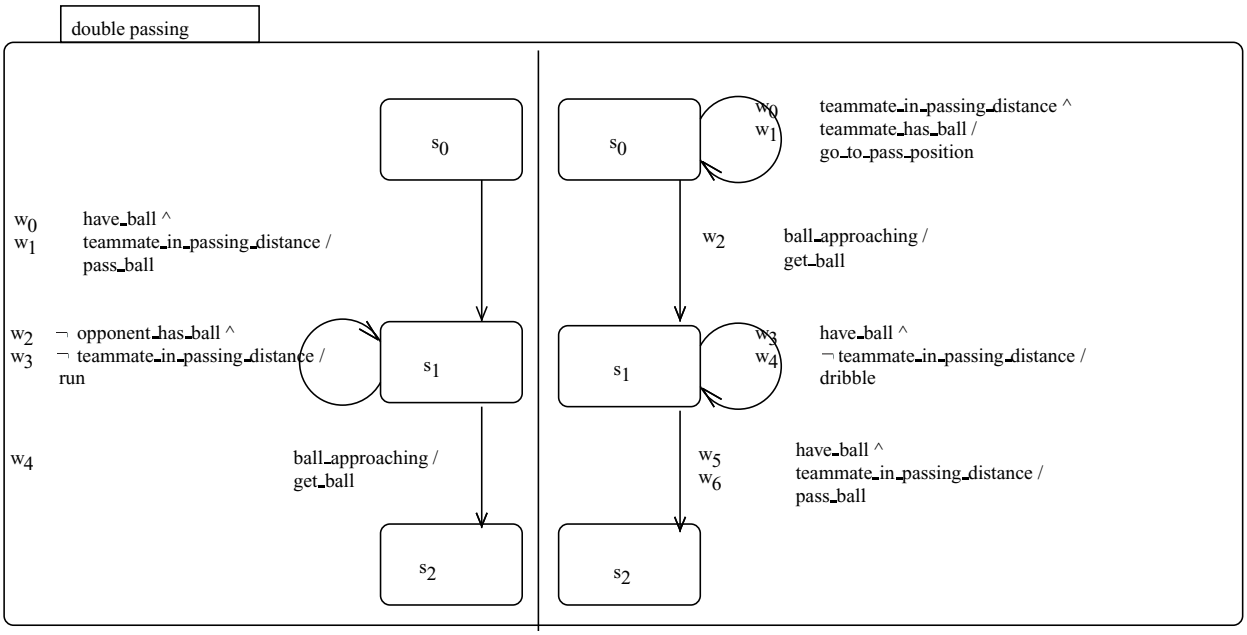


Figure 2.1: UML statechart for two agents involved in a double pass (from [58]). Two concurrent sets of states are shown together with conditions for transitions and actions that are carried out during transitions.

specifications are translated “almost automatically” into a running implementation (into Prolog), the main purposes of the modeling are the verification and formal analysis of the high level behaviors of an agent system.

The interesting notion of concurrent states (cf. fig. 2.1) allows for modeling and analyzing cooperative behaviors performed by pairs of agents. In addition, applying UML statecharts allows them to use existing graphical UML editing tools.

2.2.5 The Configuration Description Language

From the author’s point of view, the best and most advanced language for behavior specification in the related work is the *Configuration Description Language* (*CDL*) as a part of the *MissionLab* system [48, 40]. In this language, reactive stimulus-response behaviors perform the actions of a system and are implemented in a native programming language (C++). Since these primitive behaviors all fulfill a specific task autonomously and in interaction with the environment, the authors associate them with autonomous *atomic agents* as introduced in Minsky’s theory of agent societies [51].

Assemblage agents for more complex tasks are constructed by combining and coordinating subordinated agents (basic behaviors). Inside the assemblage agents, state machines are responsible for selecting one of

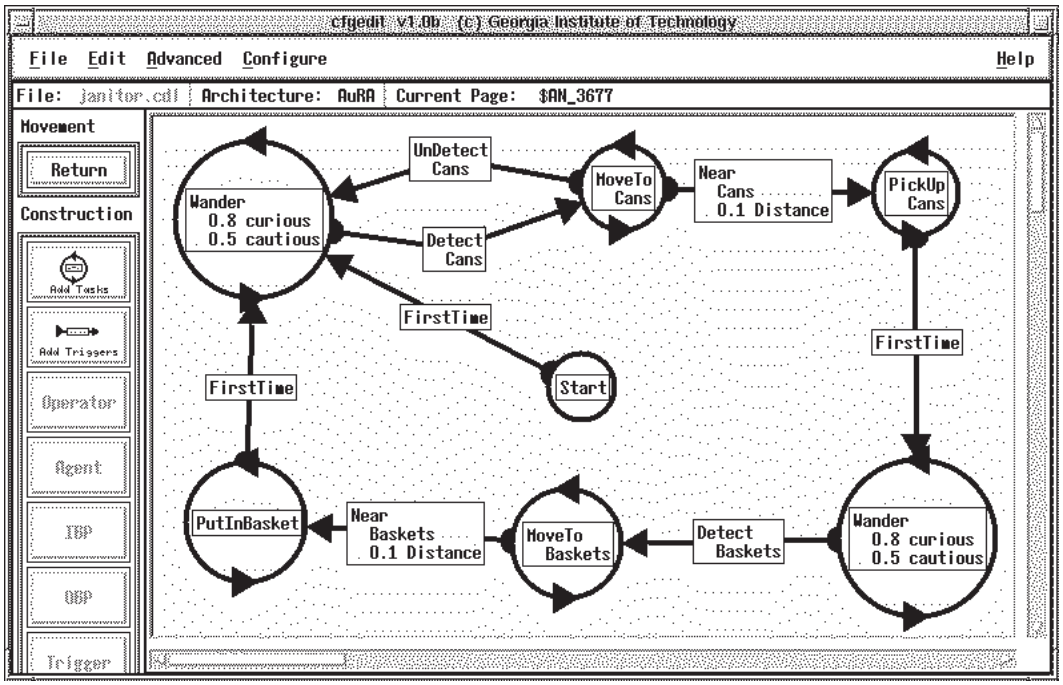


Figure 2.2: The graphical *Configuration Editor* (*CfgEdit*) as a part of the *MissionLab* toolset (from [48]). A state machine describing the task of a trash collecting robot is shown.

the subordinated behaviors. Each state in the state machine of the agent denotes a member agent which has the control over the actions of the systems while the state is active. The transitions between states are triggered by perceptual signals. Such assemblage agents can be treated as primitive behaviors to construct more complex assemblages. The possibility for constructing assemblages recursively from others allows to reuse well-designed behaviors in different contexts.

CDL is not only a language for behavior specification but also for the design of complete agent systems. Different levels of abstraction allow to bind developed solutions to different robotic platforms. However, a minor disadvantage of applying CDL could be that the agent architecture of the own system has to be largely adapted to the MissionLab system, which is often only possible when developing a new system from scratch.

The graphical *Configuration Editor* (*CfgEdit*, cf. fig. 2.2) is the most interesting feature of the system. It allows to create recursive assemblies graphically. In addition, debug tools and a robot simulator are integrated.

3 The Extensible Agent Behavior Specification Language (XABSL)

The *Extensible Agent Behavior Specification Language (XABSL)* [46] is an XML based language for behavior engineering. It simplifies the process of specifying complex behaviors and supports the design of both very reactive and long term oriented agent behaviors. It is not only a behavior modeling or description language – instead, behaviors written in *XABSL* can be transformed automatically into an intermediate code which is executed directly on a target platform using the *XabslEngine* class library. Together with the interpreter and a variety of tools for visualization and debugging, behavior developers get a complete system for behavior specification, documentation, testing, execution, and debugging. The whole *XABSL* system can be downloaded for free from the *XABSL* web site [45].

Section 3.1 describes hierarchical finite state machines for action selection as the behavior control architecture behind *XABSL*. Section 3.2 gives an overview of the *XABSL* language and section 3.3 provides a brief introduction to the language elements and the syntax. Section 3.4 deals with some technical issues related to the use of XML techniques and the tools that were developed in conjunction with *XABSL*. Section 3.5 describes the runtime system *XabslEngine*. Finally, section 3.6 relates the architecture and the language to other approaches.

3.1 Hierarchies of Finite State Machines

In *XABSL*, behavior modules (*options*) that contain state machines for decision making are ordered in a hierarchy, the *option graph*, with atomic *basic behaviors* at the leaves.

3.1.1 The Option Graph

An *XABSL* behavior specification consists of a set of behavior modules called *options* and a set of distinct simple actions (skills) called *basic behaviors*. Both options and basic behaviors can have parameters. The options are ordered in a hierarchy – complex behaviors are composed from simpler ones. Each option uses a set of other subordinated options and/or basic behaviors to realize a certain behavior.

For example in figure 3.1, the option *"grab-ball-with-head"* (a behavior for grabbing and holding the ball between the front legs and the head of an Aibo robot) is composed of the option *"approach-ball"* (a behavior for walking to the ball) and the basic behavior *"walk"* (a behavior for blind walk).

Each basic behavior and option can be used from more than one other option. This allows to reuse the same behaviors in different contexts. E.g. in figure 3.1 a few other options than *"grab-ball-with-head"* use the option *"approach-ball"*. This helps behavior developers to modularize their behaviors. In the example, only one behavior for ball approaching was developed and fine-tuned and then used by very different other options.

The option hierarchy can be seen as a rooted directed acyclic graph, called the *option graph*. The basic behaviors are the leaves (terminal nodes) of this graph. The "topmost" option (at the root of the graph) is called the *root option*. Note that in *XABSL* it is possible to specify option graphs that contain loops (and are for this reason not acyclic). But the runtime system is able to detect such loops at startup and denies work if the graph is not acyclic.

In the architecture, action selection means to activate, parameterize, and execute one of the basic behaviors. Therefore, the root option (which is always active) activates and parameterizes one of its subsequent options, this subsequent option again activates and parameterizes one of its subsequent options or basic behaviors and so on until a basic behavior is activated, parameterized, and executed. As the option graph is directed and acyclic, always exactly one of the basic behaviors is reached and executed.

In *XABSL*, a subset (sub-graph) of the options and basic behaviors which is spanned by a specially marked option, the *root option*, is called an *agent*. (As the option graph does not need to be connected completely, it is not possible to determine a single root option of the graph – *agents* mark the root options of different trees.)

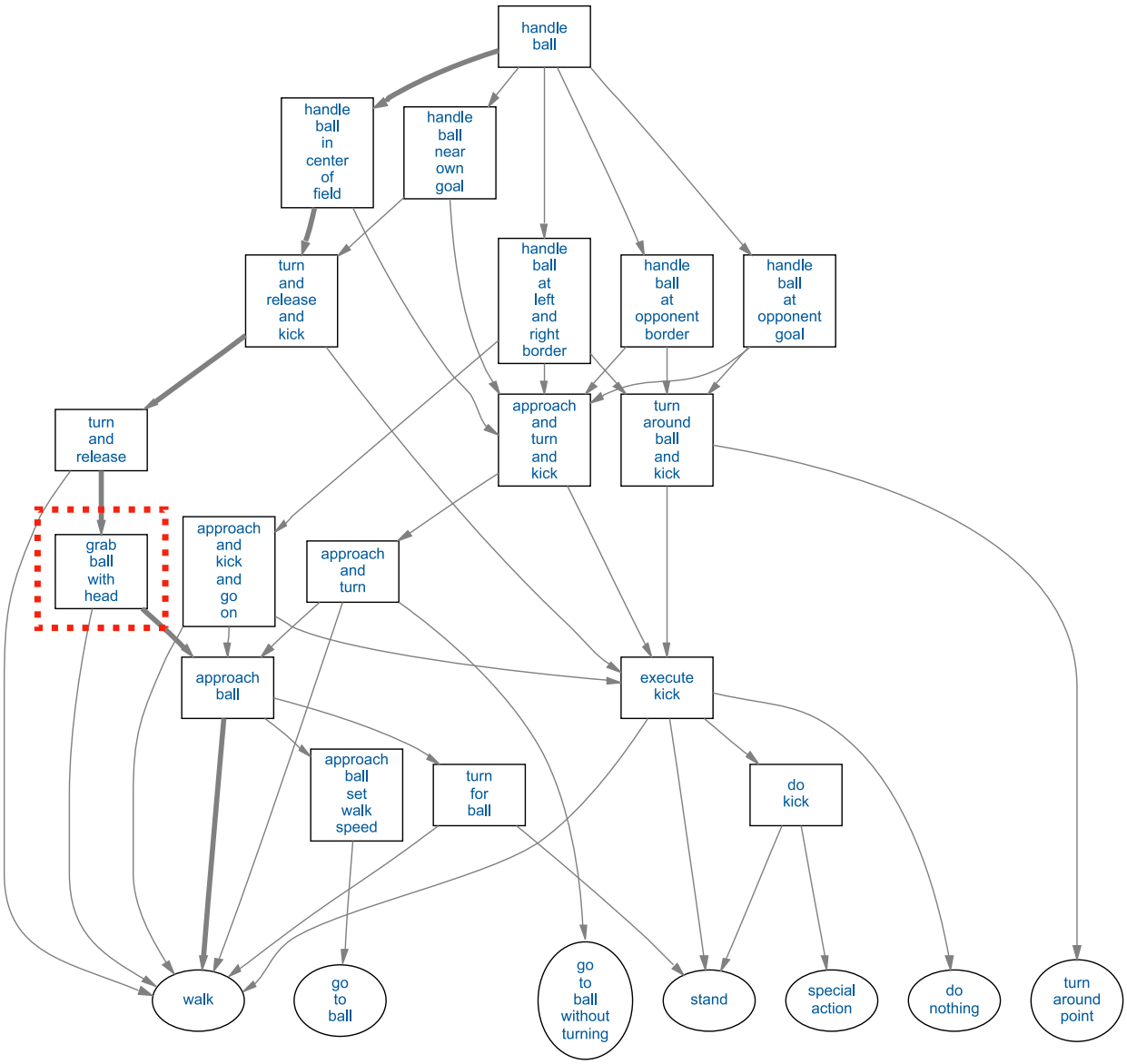


Figure 3.1: An example for an option graph from the robot soccer domain (the ball handling part of the *GermanTeam*'s soccer behaviors for the world championships 2004 in Lisbon). Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from within an option. The thick edges mark one of the many possible option activation paths. The internal state machine of option "*grab-ball-with-head*" (marked with the dashed rectangle) is shown in figure 3.2.

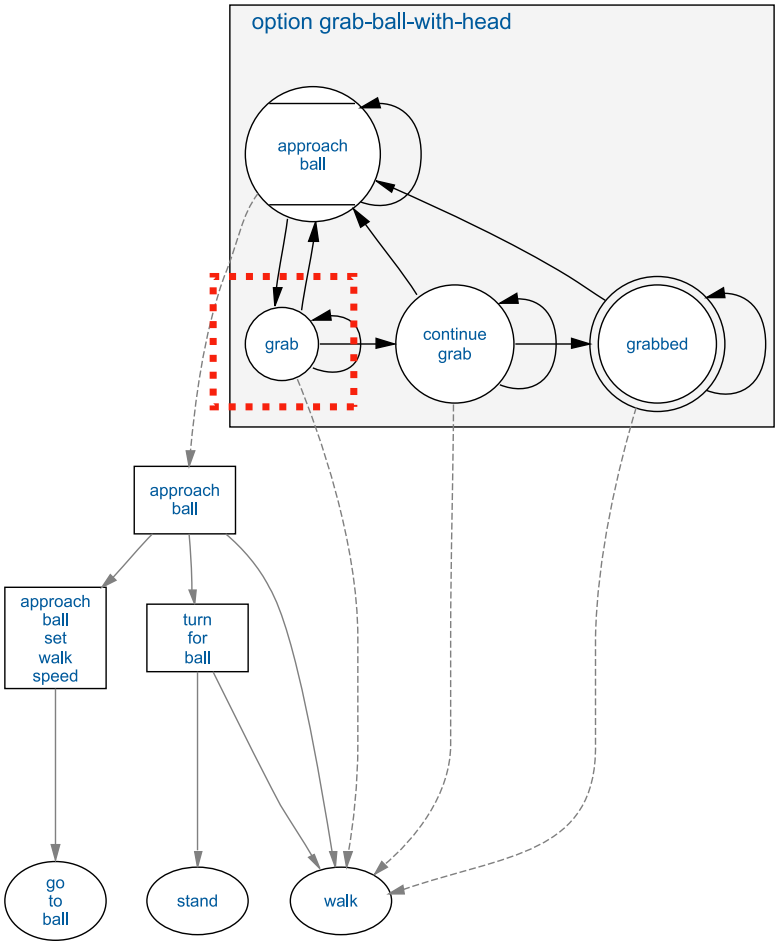


Figure 3.2: An example for an option's internal state machine (the option *"grab-ball-with-head"* from the example in figure 3.1). Circles denote states, the circle with the two horizontal lines denotes the initial state, the double circle denotes a target state. An edge between two states indicates that there is at least one transition from one state to the other. The dashed edges show which other option or basic behavior becomes activated when the corresponding state is active. The decision tree of state *"grab"* (marked with the dashed rectangle) is shown in figure 3.3.

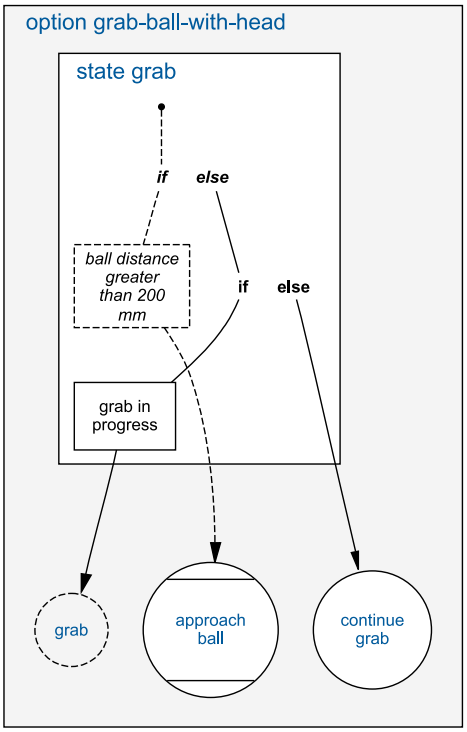


Figure 3.3: An example for a decision tree of a state (state *"grab"* of option *"grab-ball-with-head"* in figure 3.2). The leaves of the tree are transitions to other states. The dashed circle denotes a transition to the same state. The pseudo code of that decision tree is shown in figure 3.4.

3.1.2 State Machines

Within options, the activation of subordinated behaviors is done by finite state machines. Figure 3.2 shows an example of such a state machine. In each option, exactly one state is marked as the *initial state*. This state gets activated when the option becomes newly activated. An arbitrary number of states can be declared as *target states*. This allows to indicate that a behavior is finished as higher options can query whether a subsequent option reached a target state. Each state is connected to exactly one subsequent option or subsequent basic behavior. Note that more than one state can be connected to the same subsequent option or basic behavior. Always exactly one state of an option is active. This state determines, which of the subordinated behaviors is activated and how its parameters are set.

Each state has a *decision tree*, which selects a transition to either another or the same state. Figure 3.3 gives an example for such a decision tree. For the decisions, the following information can be used: Parameters passed by higher options, the world state, other sensory information, and messages from other agents. As timing is often important, it can also be taken into account how long the state and the option are already

```
if ((ball.time-since-last-seen-consecutively < 200)     // ball distance greater than 200 mm
    && (ball.consecutively-seen-time > 100)
    && (ball.seen.distance > 200)
    && (ball.seen.distance < 800)                    )
{
  transition-to-state(approach-ball);
}
else
{
  if (time-of-state-execution < 1000)    // grab in progress
  {
    transition-to-state(grab);
  }
  else
  {
    transition-to-state(continue-grab);
  }
}
```

Figure 3.4: The pseudo code of the decision tree of state *"grab"* (cf. fig. 3.3).

active. In addition, the success of a subsequent option can be tested by querying whether the subsequent option reached one of its target states.

As each state has its own decision tree, the decisions are made not only dependent on the representation of environment's state but also on the decisions that were done in the past. When the active state is taken into account, hysteresis functions between states are possible. That means if there is a transition from state A to state B for a certain condition, this condition can be different than for the transition from B to A. Thus, behaviors can be preferred once they were selected to avoid oscillations.

In the robot soccer example from figure 3.2, the option *"grab-ball-with-head"* is initially in the state *"approach-ball"*. As long as the state is active, the subsequent option *"approach-ball"* is activated with certain parameters, making the robot move towards the ball. As soon as the ball gets closer than a threshold, the decision tree of state *"approach-ball"* selects a transition to state *"grab"*. State *"grab"* becomes the active state and the subsequent basic behavior *"walk"* is executed with parameters such that the robot walks onto the ball. If it somehow happens that during that the ball gets farer away than another, the decision tree of state *"grab"* selects a transition back to state *"approach-ball"*. Otherwise, after a certain time a transition to state *"continue-grab"* is selected (cf. fig. 3.4).

3.1.3 Interaction with the Environment

To access the information about the world that is needed for decision making, symbolic representations are used. The world model of the agent system is divided into simple and non-structured information items, called the *input symbols*. In the ball grabbing example, amongst others the symbol *"ball.seen.distance"* is used to reference the distance to the seen ball in the world model.

The main actions of the agent system are controlled by the basic behaviors. It does not matter if these actions are generated completely reactively using closed sensor-actuator loops or if intermediate representations such as a world model are used in addition. In embodied agents, the basic behaviors usually control the agent's locomotion system. E.g. in the soccer behaviors of the *GermanTeam*, the basic behaviors were used to control all leg movements of the robots (walking and kicking).

Besides the execution of basic behaviors, the environment can be influenced by setting special requests, the *output symbols*. Each state within an option can set such output symbols to certain values to control perception processes or additional actuators. For instance, for the robots of the *GermanTeam*, an important actuator independent from the leg movements is the head. The output symbol *"head-control-mode"* is used to set a general mode how to move the head independent from the selected basic behavior. This mode is then used by other parts of the software to control the head movements. But also LED and sound output and messages to team mates are triggered with output symbols.

3.1.4 The Execution of the Option Hierarchy

An *XABSL* behavior implementation is always a part of a wider agent program. The surrounding software has to process the sensor readings, build up (if necessary) a world model, manage the communication to other agents, control the actuators and so on. At some point in this *sense-think-act cycle*, the program passes the control to the *XABSL* system to execute the option graph. Before, all data needed for decision making have to be up to date. Afterwards, the actions generated by the basic behaviors and the additional requests set by the output symbols have to be (processed and) sent to the actuators of the agent system.

Each time the option graph is executed, a basic behavior becomes selected and executed. The *XABSL* system has to be executed as frequent as required for the reactivity of the action system. Usually, it is called as often as new data can be obtained from the agent's main sensor. For instance on the Aibo robots of the *GermanTeam*, the *XABSL* behaviors are always executed after a newly perceived image was processed.

The execution of the option graph starts from the root option (cf. sect. 3.1.1) of the agent. The decision

tree of the active state of the root option is executed to determine the next active state, which can of course be the same as before. For the subsequent option of the active state, again the decision tree of the active state is executed and so on until the subsequent behavior of a state is a basic behavior.

Each time a decision tree activates another or the same state, the newly activated state sets the parameters of the subsequent option or basic behavior and the state's output symbols. Note that output symbols that were set during this process can be overwritten by options lower in the option graph. If an option was not active during the last execution of the option graph, the state machine is reset (the initial state is activated).

The *option activation path* (cf. fig. 3.1) follows the path from the root option to the currently activated basic behavior through all active options. As each option activates only one subsequent behavior at a time and as the graph is rooted, directed, and acyclic, such a path exists and contains no branches. The *time of option activation* is the time, how long an option was consecutively activated. This time is set to zero when an activated option was not active during the last execution of the option graph. Accordingly, the *state execution time* is the time how long the active state was consecutively activated.

The option activation path including the option activation time, active state, and state activation time for all of its options constitute the global state of an *XABSL* agent. The generated actions of the system depend on this state, the perceptions and the world model (and, if the basic behaviors have persistent states, on these states).

3.2 Behavior Specification in XML

Implementing such an architecture totally in C++ proved to be error prone and not very comfortable [15]. The source code became very large and it was quite hard to extend the behaviors. Therefore, the *Extensible Agent Behavior Specification Language* (*XABSL*) was developed to simplify the behavior engineering process.

The *XABSL* language and supporting tools are completely based on XML techniques. Figure 3.5 shows an example of an *XABSL* XML notation. The reasons to use XML instead of defining a new grammar from scratch were the big variety and quality of existing editing, validation, and processing tools, the possibility of easy transformation from and to other languages as well as the general flexibility of data represented in XML languages. The syntax and even all constraining relations between the language elements are specified in XML schema, so no other compile or validation tools than standard XSLT / XML processors are needed[1].

[1]The only exception is the check for loops in the option graph. This can not be done by validating documents against XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE symbol-and-basic-behavior-files SYSTEM "../symbol-and-basic-behavior-files.dtd">
<option xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.2
../../../../Tools/Xabsl2/xabsl-2.2/xabsl-2.2.option.xsd" name="grab-ball-with-head" initial-state="approach-ball">
  &ball-symbols;
  &head-and-tail-symbols;
  &motion-request-symbols;
  &special-action-symbols;
  &strategy-symbols;
  &robot-state-symbols;
  &common-basic-behaviors;
  &simple-basic-behaviors;
  &options;
    <common-decision-tree>
    <if>
      <condition description="ball distance greater than 200 mm">
        <and>
          <less-than>
            <decimal-input-symbol-ref ref="ball.time-since-last-seen-consecutively"/>
            <decimal-value value="200"/>
          </less-than>
          <greater-than>
            <decimal-input-symbol-ref ref="ball.consecutively-seen-time"/>
            <decimal-value value="100"/>
          </greater-than>
          <greater-than>
            <decimal-input-symbol-ref ref="ball.seen.distance"/>
            <decimal-value value="200"/>
          </greater-than>
          <less-than>
            <decimal-input-symbol-ref ref="ball.seen.distance"/>
            <decimal-value value="800"/>
          </less-than>
        </and>
      </condition>
      <transition-to-state ref="approach-ball"/>
    </if>
  </common-decision-tree>
  ...
  <state name="grab">
    <subsequent-basic-behavior ref="walk">
      <set-parameter ref="walk.type">
        <constant-ref ref="walk-type.normal"/>
      </set-parameter>
      <set-parameter ref="walk.speed-x">
        <decimal-value value="200"/>
      </set-parameter>
      <set-parameter ref="walk.speed-y">
        <decimal-value value="0"/>
      </set-parameter>
      <set-parameter ref="walk.rotation-speed">
        <multiply>
          <decimal-input-symbol-ref ref="ball.seen.angle"/>
          <decimal-value value="2"/>
        </multiply>
      </set-parameter>
    </subsequent-basic-behavior>
    <set-output-symbol ref="head-control-mode" value="head-control-mode.catch-ball"/>
    <set-output-symbol ref="ball.handling" value="handling-the-ball"/>
    <decision-tree>
      <if>
        <condition description="grab in progress">
          <less-than>
            <time-of-state-execution/>
            <decimal-value value="1000"/>
          </less-than>
        </condition>
        <transition-to-state ref="grab"/>
      </if>
      <else>
        <transition-to-state ref="continue-grab"/>
      </else>
    </decision-tree>
  </state>
  ...
</option>
```

Figure 3.5: An example for an *XABSL* XML notation: a source code fragment for the state *"grab"* (cf. fig. 3.3) of option *"grab-ball-with-head"* (cf. fig. 3.2).

Many XML Editors are able to check whether an *XABSL* document is valid at runtime. A high validation and compile speed results in short change-compile-test cycles.

Standard XSLT transformations are used to compile *XABSL* documents to an intermediate code for the runtime system and to generate extensive documentations. Note that the figures 3.1, 3.2, 3.3, and 3.4 were generated automatically from the XML source in figure 3.5.

An aftereffect of this restriction to standard XML technologies and tools is that the language had to be adapted to existing tools to some extend. For example, some constructs had to be introduced only for the compatibility with the used XML editor. And, which is also not typical for a programming language, there is a relatively strict distribution of language elements onto different file types, which is required for efficient processing of the data (in previous versions of *XABSL*, the complete specification of the behaviors was in only one file, which made editing very slow).

Agent behavior specifications based on the architecture introduced in the previous section can be completely described in *XABSL*. There are language elements for options, their states, and their decision trees. Boolean logic ($||$, $\&\&$, !, $==$, $!=$, $<$, $<=$, $>$, and $>=$), simple arithmetic operators ($+$, $-$, $*$, $/$, and $\%$), and conditional decimal expressions (comparable to the ANSI C question mark operator, $a\ ?\ b : c$) can be used for the specification of decision trees and parameters of subsequent behaviors. 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 conditions, a symbol has to be defined. This makes the *XABSL* framework independent from specific software environments and platforms. An example:

```
<decimal-input-symbol name="ball.x" measure="mm"
  description="The absolute x position on the field"/>
<decimal-input-symbol name="utility-for-dribbling"
  measure="0..1" description="Utility for dribbling"/>
<boolean-input-symbol name="goalie-should-jump-right"
  description="A ball rolls along to the right"/>
```

Schema and is therefore checked by the runtime system at startup.

The first symbol *"ball.x"* simply refers to a variable in the world state of the agent system, *"utility-for-dribbling"* stands for a member function of an utility analyzer and *"goalie-should-jump-right"* represents a complex predicate function that determines whether a fast moving ball is headed to the right portion of the own goal. In options, these symbols then can be referenced.

The developer may decide whether to 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.

As the *basic behaviors* are written in C++, prototypes and parameter definitions have to be specified in an *XABSL* document so that states can reference them.

3.3 The XABSL Language

This section gives a brief introduction to the syntax and the semantics of the *XABSL* language. Thereby, the formal structure of the grammar is, as usual in the XML world, displayed with syntax diagrams (e.g. fig. 3.6) instead of textual representations such as EBNF or others. A complete language reference can be found at the *XABSL* web site [45].

3.3.1 Symbols, Basic Behaviors and Option Definitions

Symbols, basic behaviors, and option definitions are referenced from inside options. In order that it can be checked whether a referenced symbol (or option parameter etc.) exists, they all have to be declared in definition files (comparable to header files in C++).

First, there are definition files for symbols. There can be many of them for grouping symbols thematically. The element *"symbols"* is the root element of such a symbol file (cf. fig. 3.6). *XABSL* has six different symbol types that can be declared in arbitrary order inside a symbols element: A *"boolean-input-symbol"* represents a symbol for a Boolean, and a *"decimal-input-symbol"* a symbol for a decimal variable or function (the *XabslEngine* uses the data type double for decimal values). Besides the attribute *"name"*, which is the id of the symbol and which is referenced from inside options, it has additional attributes that are needed for the generation of the HTML documentation. A *"decimal-input-function"* is a prototype for a parameterized decimal function. Each parameter of a function is defined in a separate *"parameter"* child element. The element *"enumerated-input-symbol"* represents a symbol for an enumerated variable or function. Each enumerated item is defined in a single *"enum-element"* child element. Output symbols

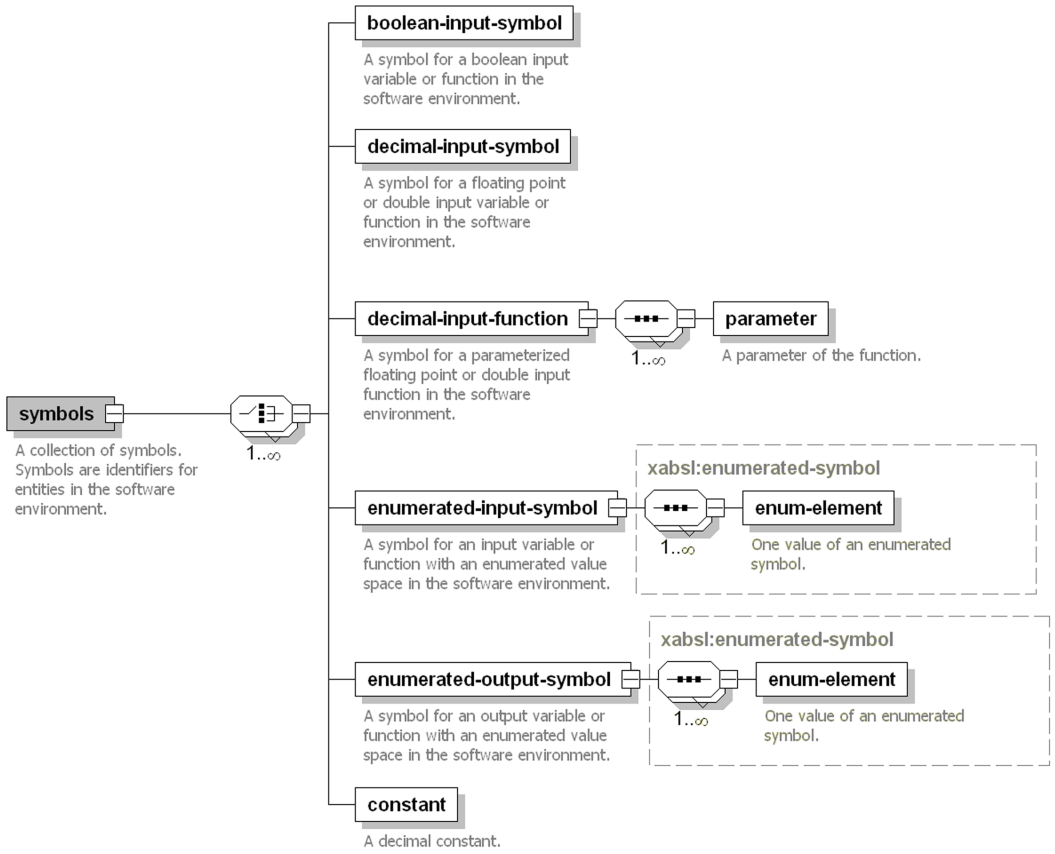


Figure 3.6: The syntax of the element *"symbols"*.

are declared with *"enumerated-output-symbol"*, like the *"enumerated-input-symbol"* element with *"enum-element"* child elements. The element *"constant"* defines a decimal constant.

Basic behaviors are written in C++. Nevertheless, in basic behavior files, a prototype has to be declared for each of them. The element *"basic-behaviors"* (cf. fig. 3.7) is the root element of such a file and has to have at least one child element of the type *"basic-behavior"*, which defines a prototype for a basic behavior. Optionally it has *"parameter"* child elements which declare a parameter that can be passed to the corresponding basic behavior written in C++.

Every option is encapsulated in an own file. To be able to validate a single option (e. g. the existence of a referenced subsequent option), there must be prototypes for all other options. Therefore, in each *XABSL* agent behavior specification a file named "options.xml" has to exist. It has an *"option-definitions"* (cf. fig. 3.7) root element. Inside, *"option-definition"* elements define a prototype for an option. As the *"basic-*

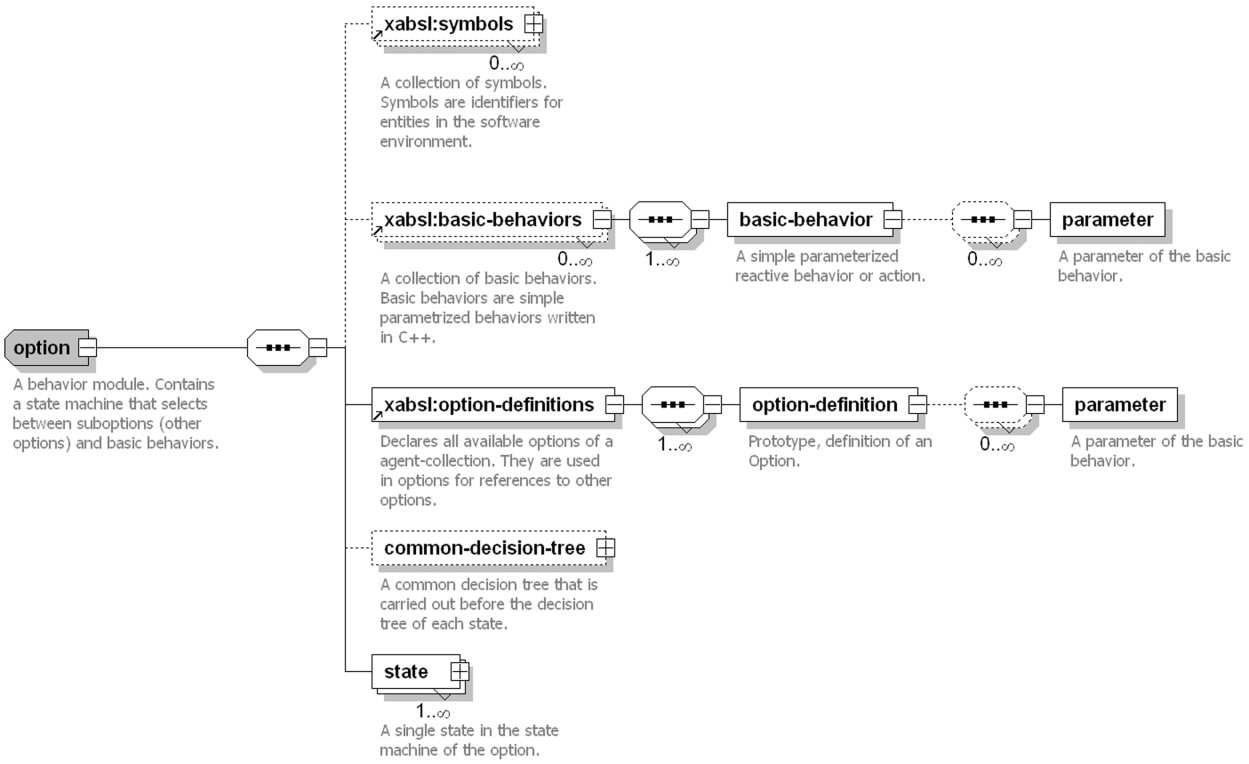


Figure 3.7: The syntax of the element *"option"*.

behavior" element, it can have *"parameter"* child elements that specify parameters of an option.

3.3.2 Options and States

The root element of an option file is the *"option"* element (cf. fig. 3.7). Inside that, the files for all referenced symbol definitions and basic behavior and option prototypes are included using a DTD include mechanism (cf. sect. 3.4).

After the included *"symbols"*, *"basic-behaviors"*, and *"option-definitions"* child elements, a *"common-decision-tree"* child element can follow. This is a decision tree which is carried out before the decision tree of the active state. If no condition of the common decision tree proves to be true, the decision tree of the active state is carried out. This can be used to reduce the complexity of implementation when the conditions for a transition are same in each state. If the common decision tree uses expressions that are specific for a state (*"time-of-state-activation"* or *"subsequent-option-reached-target-state"*), these expressions refer to the state that is currently active. The child elements of a *"common-decision-tree"* are the same as in the normal decision tree of a state, which is explained later in this section.

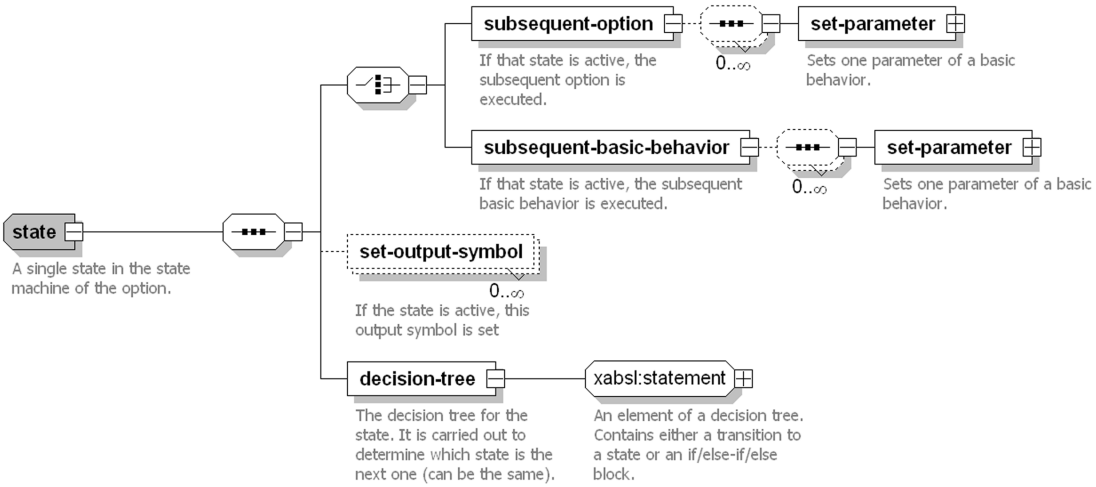


Figure 3.8: The syntax of the element *"state"*.

Followed by the optional *"common-decision-tree"*, each option has to have at least one *"state"* child element, which represents a single state of an option's state machine (cf. fig 3.8). Its first child element is either a *"subsequent-option"* or a *"subsequent-basic-behavior"*, determining which subsequent behavior is executed when this state is active. If the referenced option or basic behavior has parameters, these can be set with *"set-parameter"* child elements. If a state does not set all parameters of a subsequent behavior, the execution engine sets the remaining parameters to zero. The child element of the *"set-parameter"* element is a decimal expression, which are described later in this section.

After the definition of the subsequent behavior, output symbols can be set by inserting *"set-output-symbol"* child elements. Note that the state machine is carried out first and only the then active state can set these symbols. It may happen that an option which becomes activated lower in the option graph overwrites an output symbol. The output symbols are only applied to the software environment when the option graph was executed completely.

Each state has a decision tree. The task of this decision tree is to determine a transition to a following state (which can be the same state). Consequently, the leaves of a decision tree are transitions to other states. The element *"decision-tree"* itself is of the type *"statement"* (cf. fig. 3.9). A *"statement"* can be either an if, else-if, else block or a transition to a state. The *"transition-to-state"* element represents a transition to another state.

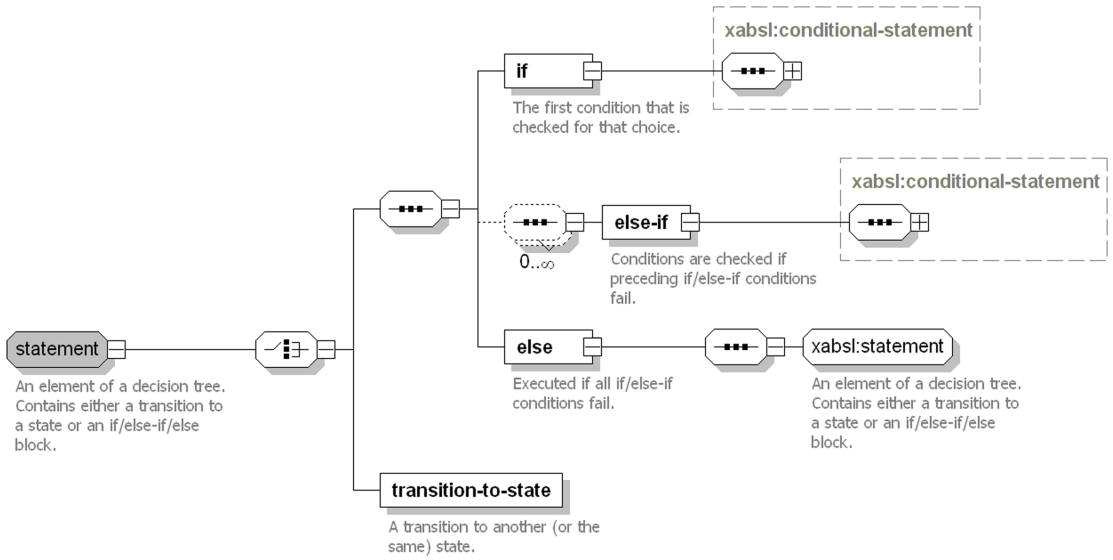


Figure 3.9: The syntax of the group *"statement"*. Amongst others, the element *"decision-tree"* is of this type.

An if, else-if, else block consists of an *"if"*, optional *"else-if"* and an *"else"* element. The *"if"* and the *"else-if"* elements both have a *"condition"* child element and a statement which is executed if the condition is true. The statement itself is again either a if/else-if/else block or a transition to a state, which allows for complex nested expressions. The *"condition"* element has a Boolean expression (cf. next section) as a child element.

3.3.3 Boolean and Decimal Expressions

A *"boolean-expression"* can be one of the elements shown in figure 3.10. A *"boolean-input-symbol-ref"* references a Boolean input symbol. The element *"enumerated-input-symbol-comparison"* compares the value of an enumerated input symbol with a given enumerated value. The elements *"and"* and *"or"* represent the Boolean && and || operators and have at least two *"boolean-expression"* child elements. In contrast, *"not"* has only one *"boolean-expression"* child element and represents the Boolean ! operator.

The elements *"equal-to"*, *"not-equal-to"*, *"less-than"*, *"less-than-or-equal-to"*, *"greater-than"*, and *"greater-than-or-equal-to"* are the ==, ! =, <, <=, > and >= operators. They all have two *"decimal-expression"* child elements, which are described below.

The expression *"subsequent-option-reached-target-state"* is true when the subsequent behavior of the state is an option and when the active state of the subsequent option is marked as a target state. Otherwise this statement is false. It can be used to give a feed-back to higher options that a behavior is finished.

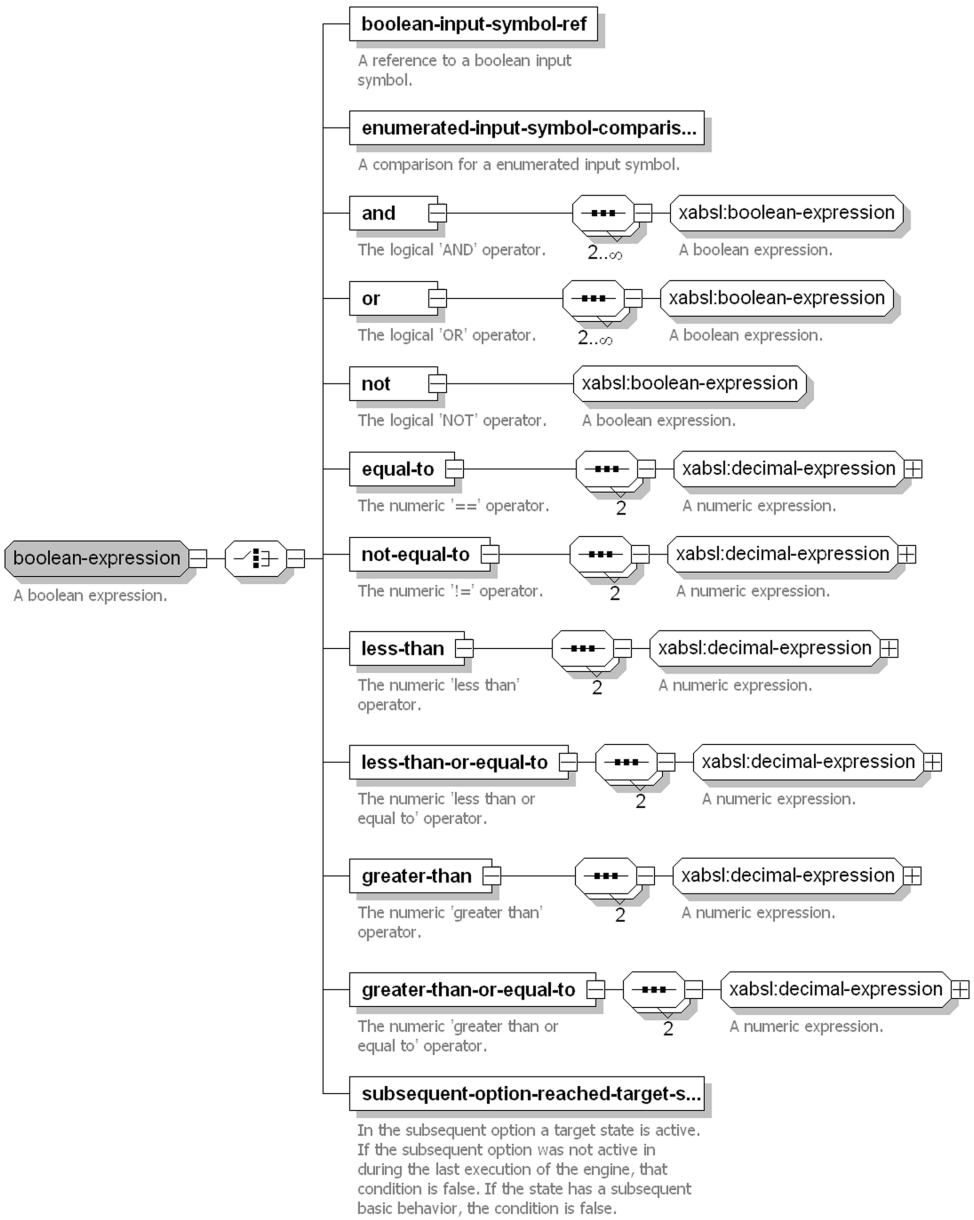


Figure 3.10: The syntax of the group *"boolean-expression"*. Elements from this group are used inside conditions of decision trees.

Elements from the *"decimal-expression"* group (cf. fig. 3.11) can be used inside some Boolean expressions and for the parameterization of subsequent behaviors.

A *"decimal-input-symbol-ref"* references a decimal input symbol. A *"decimal-input-function-call"* represents a call to a decimal input function. For each parameter of the function, a *"with-parameter"* element must be inserted. If a parameter is not set, the executing engine sets the parameter to zero.

The element *"with-parameter"* has a child element from the *"decimal-expression"* group.

A *"constant-ref"* references a constant which was defined in a *"symbols"* collection, a *"decimal-value"* is a simple decimal value, e. g. *"3.14"*, and *"option-parameter-ref"* references a parameter of the option.

The elements *"plus"*, *"minus"*, *"multiply"*, *"divide"*, and *"mod"* stand for the arithmetic $+$, $-$, $*$, $/$ and $\%$ operators. They all have two child elements from the *"decimal-expression"* group.

The element *"time-of-state-execution"* can be used to query how long the state has been already active. This time is reset when the state was not active during the last execution of the engine. Note that it may happen that the option activation path above the current option changes without this time being reset (it is only important that the option and the state were active during the last execution of the engine). Analogical, element *"time-of-option-execution"* represents the time the option has already been active. This time is reset if the option was not active during the last execution of the engine. It may also happen here that the option activation path above the current option changes without this time being reset.

The statement *"conditional-expression"* works such as an ANSI C question mark operator. A *"condition"* which has a *boolean-expression* child element is checked. If the condition is true, the decimal expression *"expression1"*, otherwise *"expression2"* is returned. It is mainly used to set parameters of subsequent behaviors (which have to be decimal) dependent on a condition.

3.3.4 Agents

The file "agents.xml" is the root document of an *XABSL* behavior specification. It includes all the options and defines agents. Figure 3.12 shows the structure of the *"agent-collection"* element. It has *"title"*, *"platform"*, and *"software-environment"* elements that are only used for generating the HTML documentation.

With an *"agent"* element, an agent is declared by referencing a root option from the set of all options. After the definition of the agents and the included option prototypes, all options that are used by the agents and all options that are referenced from other options used have to be included inside the *"options"* element using XInclude.

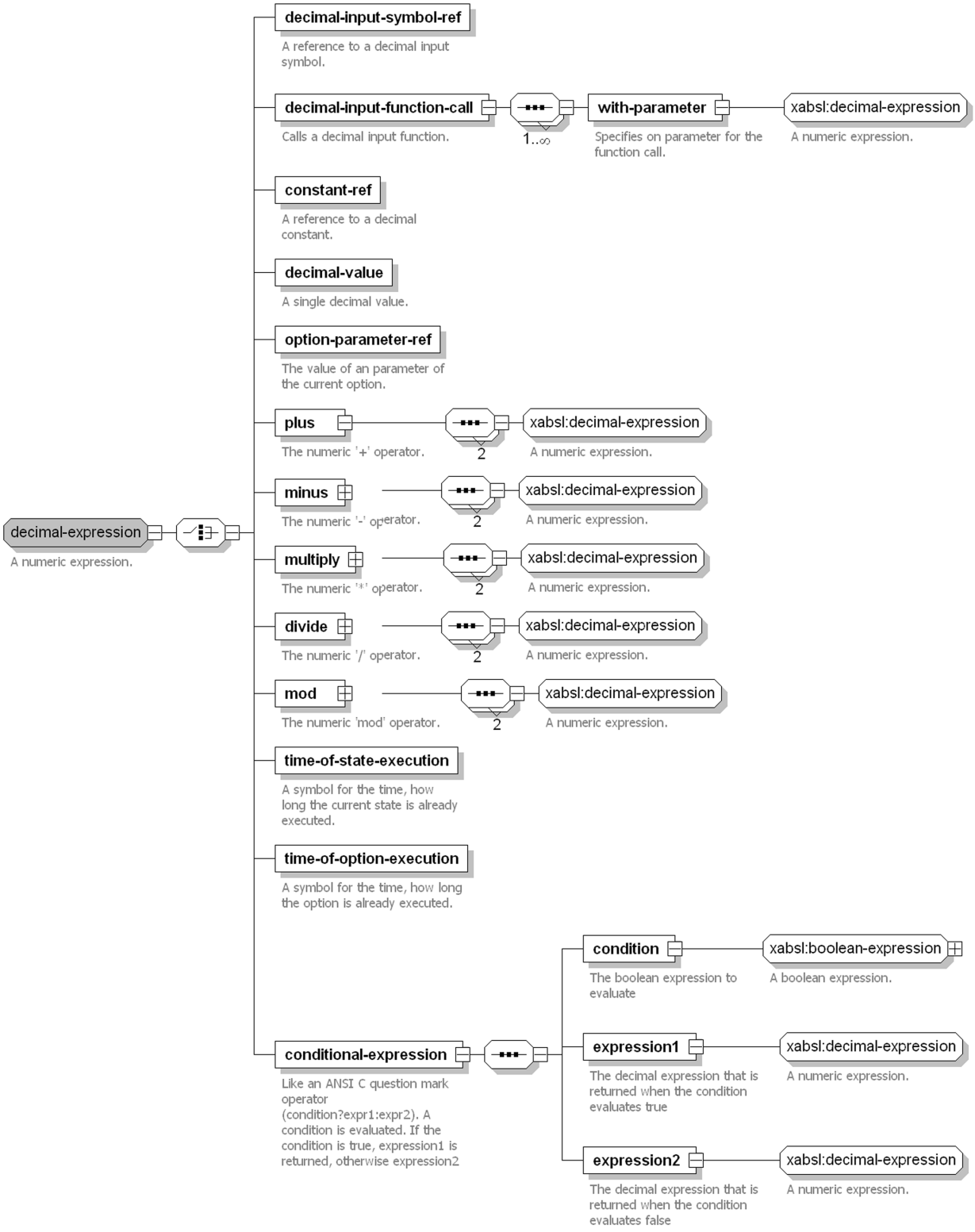


Figure 3.11: The syntax of the group *"decimal-expression"*.

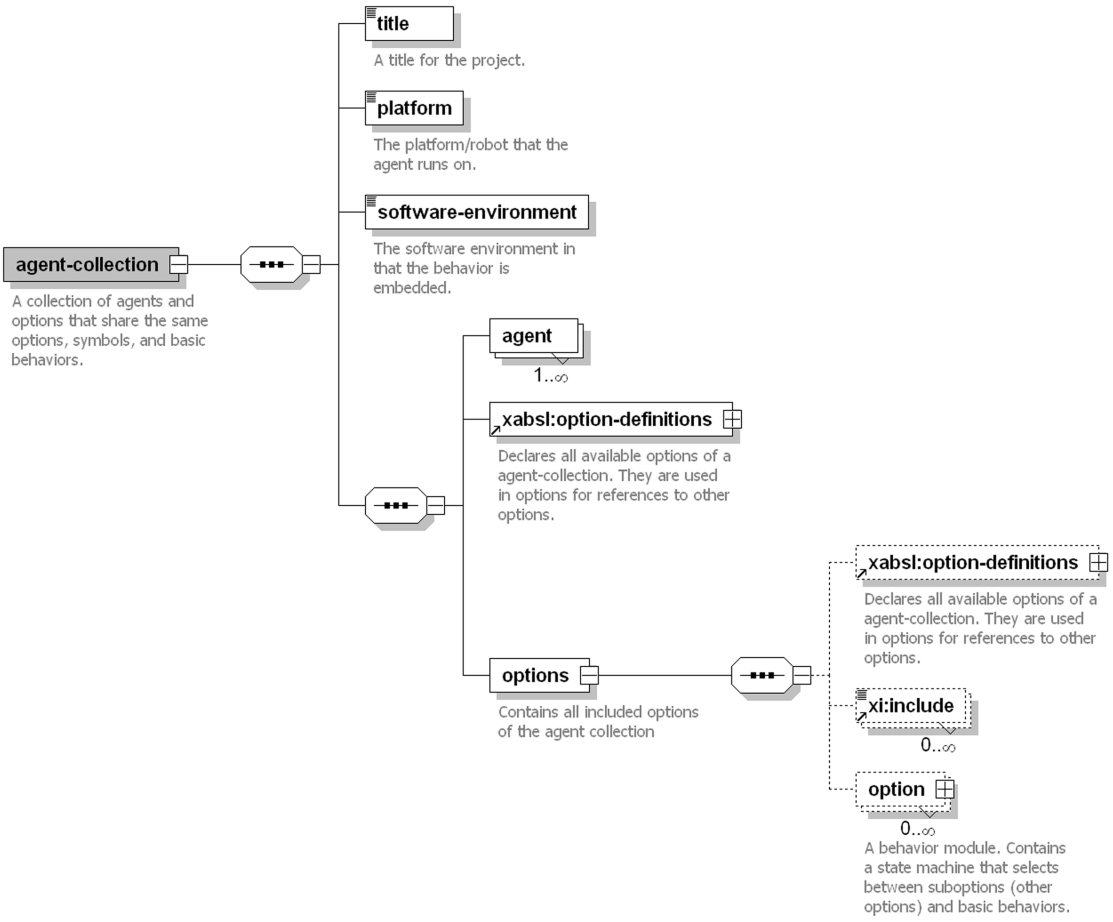


Figure 3.12: The syntax of the element *"agent-collection"*.

3.4 Mechanisms and Tools

XABSL is an *XML 1.0* [12] dialect that is specified in *XML Schema* [24]. Schemas are used instead of DTDs as only they allow to specify complex identity constraints. For instance, for all decimal input symbols there is a *key* defined which guarantees that the names of the symbols are unique. If inside an option such a decimal input symbol is referenced, a *key reference* assures that the referenced symbol exists in the key.

An *XABSL* agent behavior specification is distributed over many files, which helps the behavior developers to keep an overview over larger agents and to work in parallel. The XML schemas for all the different file types can be found at the *XABSL* web site [45].

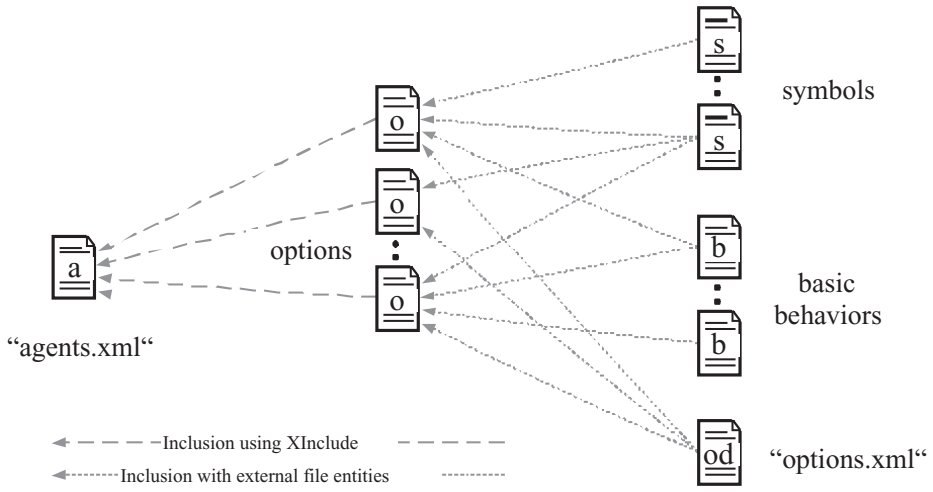


Figure 3.13: Different file types of an *XABSL* specification and include mechanisms.

3.4.1 File Types and Inclusions

Figure 3.13 shows the different file types that are part of an *XABSL* agent behavior specification. Symbol files contain the definitions of symbols, basic behavior files prototypes for basic behaviors and their parameters, and option files contain a single option. The file “options.xml” defines prototypes for each option and its parameters. The file “agents.xml” includes all the option files and defines the agents and their root options.

Two mechanisms for including one XML file into another are used. When using *External file entities*, a code block, e. g. the file “my-symbols.xml” is defined as an external file entity inside a DTD. At the correct position in the code it is inserted by for instance *&mySymbols;*. Most XML editors support this mechanism. It allows checking the validity of an option inside the XML editor. The disadvantage is that no cascading inclusions are possible.

With *XInclude* [50] a file is directly included into another one with a statement such as this: *<xinclude href=”another-file.xml”/>*. An XInclude processor later resolves these includes for further processing. The disadvantage is that most XML editors do not resolve XInclude statements for validation.

3.4.2 Document Processing

Standard XSLT [17] transformations are used to generate three types of documents from *XABSL* source documents: an *intermediate code* which is executed by the *XabslEngine*, *debug symbols* containing the names of all named elements, and an extensive HTML-documentation containing SVG-charts for each agent, option, and state.

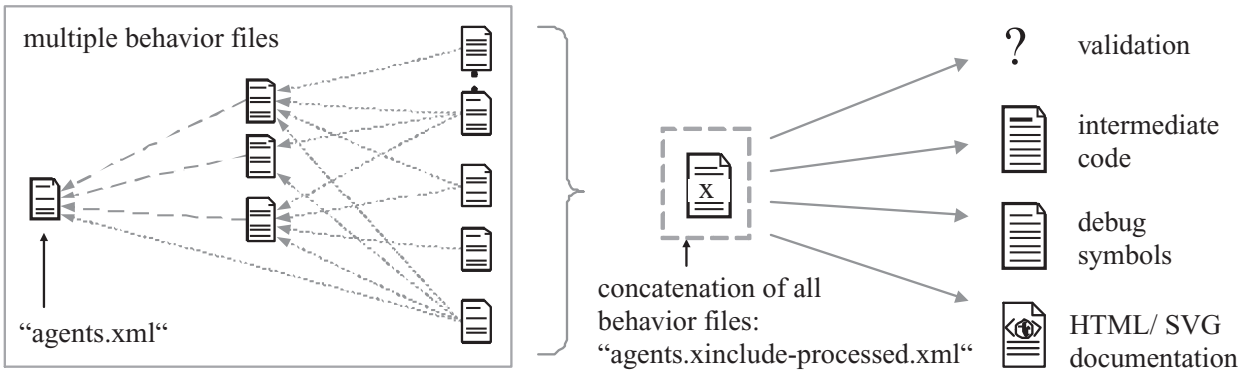


Figure 3.14: Document generation in *XABSL*

The run-time system *XabslEngine* uses an intermediate code instead of parsing the *XABSL* XML files directly, thus no XML parser is needed. (On many embedded computing platforms XML parsers are not available due to resource and portability constraints.)

The generated debug symbols contain the names of all options, basic behaviors, parameters, and symbols. They make it possible to implement platform and application dependent debugging tools for monitoring option and state activations as well as input and output symbols. For instance, the *Xabsl2 Behavior Tester Dialog* (cf. fig. 3.16) was integrated into the *RobotControl* application, the general debug tool of the *GermanTeam*.

The HTML documentation helps the developers to understand what their behaviors do. Almost all information specified in the XML files is clearly visualized, there are SVG charts for each option graph, state machine, and decision tree. As it would have been nearly impossible to generate these charts directly with native XSLT transformations (it is very difficult to place nodes and edges such that there is little overlapping), the "dot" tool of the AT&T Graphviz [26, 9] graph drawing suite was used. This program takes structural descriptions of the graphs as input and renders charts from it, ensuring a good layout and little overlappings between objects. As an XML wrapper for the input language of the "dot" tool, the *Dot Markup Language* (DotML) [44] was developed. Note that the figures 3.1, 3.2, and 3.3 were generated automatically from *XABSL* documents with DotML and "dot".

Figure 3.14 shows how all the different documents are generated. Because an *XABSL* agent behavior specification is distributed over many XML files, firstly, all these files are concatenated into a single big file "agents.xinclude-processed.xml". Then this file is validated against the *XABSL* schema. If that was success-

ful, the XSLT style sheet “generate-intermediate-code.xsl” is applied to “agents.xinclude-processed.xml” to generate the intermediate code. The debug symbols are created with “generate-debug-symbols.xsl”. Similar to the *XABSL* behaviors, the generated documentation is also distributed over many files. To increase the compile speed, only for the changed *XABSL* source files the documentation pages are rebuilt. Therefore, 13 different XSLT style sheets exist for the documentation generation.

For the correct call of all the different XSLT style sheets and DotML scripts, a complex Makefile was developed, which is described in detail on the *XABSL* web site [45].

3.5 The XabslEngine Class Library

The *Xabsl2Engine* is the *XABSL* runtime system. It is written in plain ANSI C++ [23] and does not use any extensions such as the STL [55]. It is platform and application independent and can be easily employed on any robotic platform. To run the engine in a specific software environment, only two classes (for file access and error handling) have to be derived from abstract classes.

The engine parses and executes the intermediate code that was generated from *XABSL* documents. It links the symbols from the XML 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 has to be registered to the engine. While options and their states are represented in XML, basic behaviors are written in C++. They have to be derived from a common base class and to be registered at the engine. The engine provides extensive debugging interfaces for monitoring the activation of options and states, the values of the symbols, and the parameters of options and basic behaviors. Instead of executing the engine from the root option, single options and basic behaviors can be tested separately.

A complete API documentation of the class library is available at the *XABSL* web site [45].

3.5.1 Running the Xabsl2Engine on a Specific Target Platform

As the class library is application and platform independent, message and error handling functions as well as file access routines have to be implemented externally.

First, one has to declare a message and error handling class that is derived from *Xabsl2ErrorHandler*. This class has to implement the *printMessage()* and *printError()* function. The engine uses that class to state errors and to raise error messages. The Boolean variable “*errorsOccurred*” can be used to determine whether errors occurred during the creation or execution of the engine.

Afterwards, a class that gives the engine a read access to the intermediate code has to be derived from *Xabsl2InputSource*. The code does not inevitably have to be read from a file, but can also be read from a memory region or any other stream. The pure virtual functions *open()*, *close()*, *readValue()*, and *readString()* have to be implemented.

The intermediate code contains comments (*//*...) that have to be skipped by the read functions:

```
// multiply (6)
6
// decimal value (0): 52.5
0 52.5
// reference to decimal symbol (1) ball.y
1 13
```

The comments have to be treated as in C++ files (new line ends a comment). In the example only "*6 0 52.5 1 13*" has to be read from the file.

Finally, a static function that returns the system time in milliseconds has to be defined, e.g.: *static unsigned long getSystemTime()* .

3.5.2 Registering Symbols and Basic Behaviors

After creating an instance of the *Xabsl2Engine* by passing a reference to an error handler derivate and a pointer to the time function as parameters, all the symbols and basic behaviors can be registered at the engine. Note that this has to be done before the option graph is created.

As the behaviors written in *XABSL* use symbols to interact with the software environment of the agent system, for each of these symbols the corresponding variable or function has to be registered to the engine. The following example binds the variable *aDoubleVariable* to the symbol "*a-decimal-symbol*" which was defined in the *XABSL* agent behavior specification:

```
pMyEngine->registerDecimalInputSymbol("a-decimal-symbol",
                                      &aDoubleVariable);
```

If the value of the symbol is not represented by a variable but by a function, this function has to be registered at the engine. Moreover, this function has do be defined inside a class which is derived from *Xabsl2FunctionProvider*:

```
class MySymbols : public Xabsl2FunctionProvider
{
public:
  double doubleReturningFunction() { return 3.7; }
};
...
MySymbols mySymbols;

pMyEngine->registerDecimalInputSymbol("a-decimal-symbol",
  &mySymbols, (double (Xabsl2FunctionProvider::*)())
  &MySymbols::doubleReturningFunction);
```

The registration of all other symbol types works in a similar way.

All basic behaviors are derived from the class *Xabsl2BasicBehavior* and have to implement the pure virtual function *execute()*. The name of the basic behavior has to be passed to the constructor of the base class. Furthermore, the parameters of the basic behavior have to be declared as members of the class and has to be registered using *registerParameter(..)*. Afterwards, an instance has to be registered to the engine with the *registerBasicBehavior(..)* function for each basic behavior class.

3.5.3 Creating the Option Graph and Executing the Engine

After the registration of all symbols and basic behaviors, the intermediate code can be parsed using the *createOptionGraph(..)* function.

If the engine detects an error during the execution of the option graph, the error handler is invoked. This can happen if the intermediate code contains a symbol or a basic behavior that was not registered before. By using the *Xabsl2ErrorHandler* member variable *errorsOccured*, it can be checked whether the option graph was created successfully or not.

If no errors occurred during the creation, the engine can be executed with *execute()*. This function executes the option graph once at a time. Starting from the selected root option, the state machine of each option is carried out to determine the next active state. After that, the state machine for the subsequent option of this state is carried out again and again until the subsequent behavior is a basic behavior, which is executed then, too. Finally, the output symbols that were set during the execution of the option graph become applied to

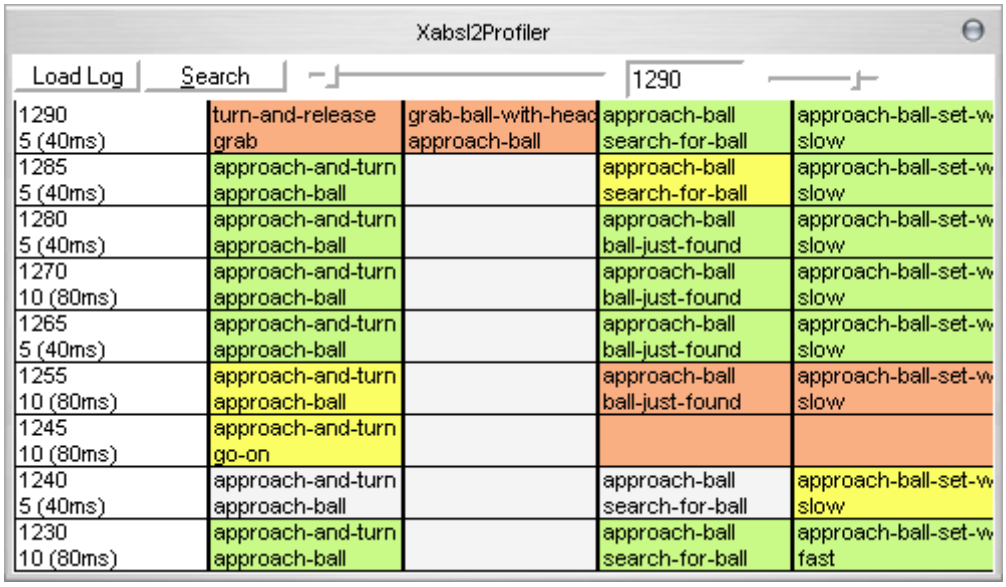


Figure 3.15: The *Xabsl2 Profiler* allows to analyze changes in the behaviors over time. Each line reports a change in the state of the *XABSL* system. In the left column, a timestamp and the number of frames with no change of state is displayed. The other columns show the corresponding option and state activations on "levels" of the option graph (each option was automatically assigned to such a level for better visualization). A red cell indicates that another option was activated on a certain level, yellow stands for a state change, and green means that the parameters of a subsequent behavior changed.

the software environment.

In the *execute()* function the execution starts from the selected root option, which in the beginning is the root option of the first agent. The agent can be switched using the function *setSelectedAgent(..)*.

3.5.4 Debugging Interfaces

The engine provides rich debugging interfaces that can be used to develop monitoring and debugging tools.

Instead of executing the option graph with *execute()*, single basic behaviors or options can be parameterized and executed separately. There is a number of functions to trace the current state of the option graph, the option activation path, the option parameters, and the selected basic behavior. For tracing the values of symbols, the engine provides access to the symbols stored. Enumerated output symbols can also be set manually for testing purposes. Note that this has to be done after the option graph was executed. The changes are applied to the software environment by using the function *setOutputSymbols()*.

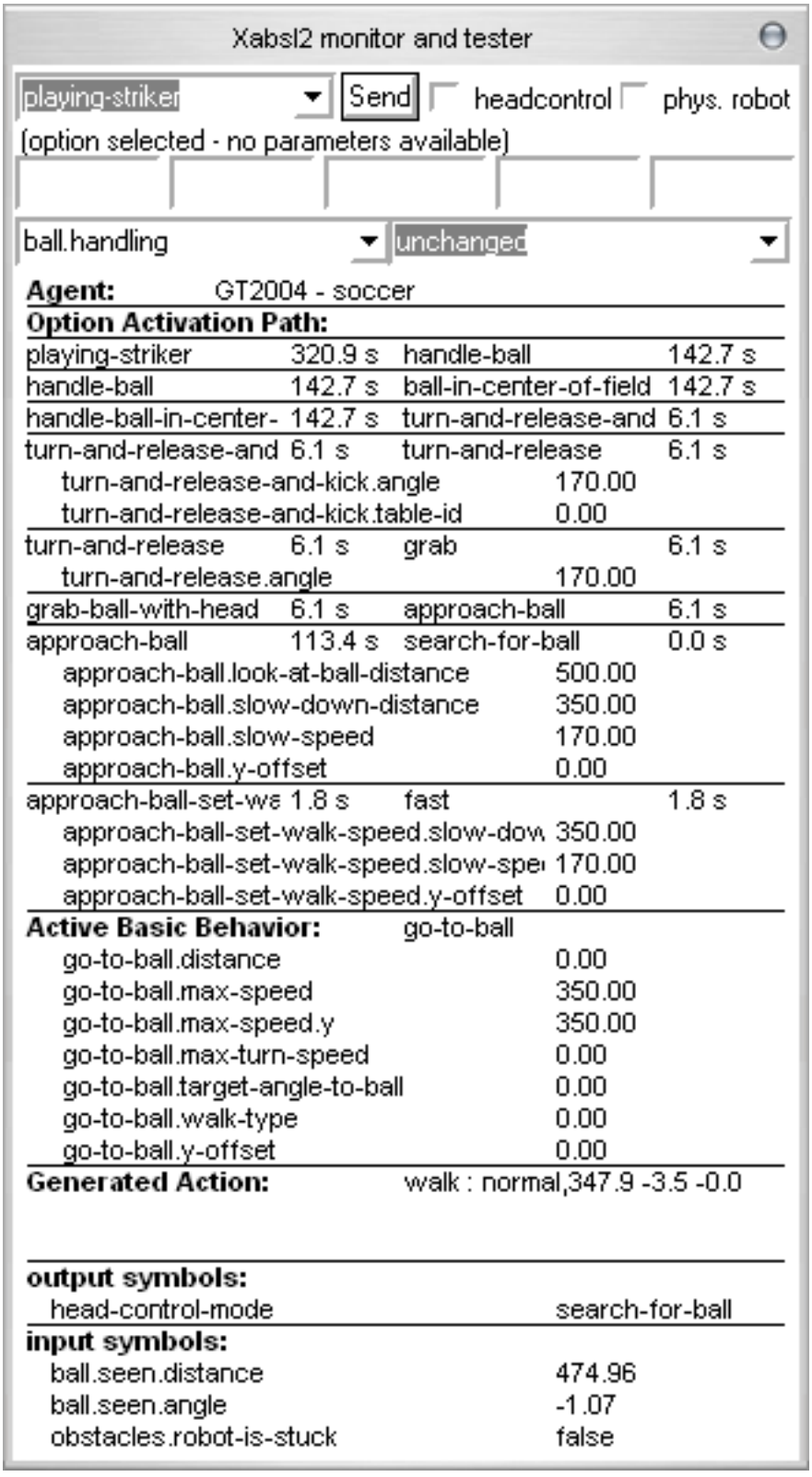


Figure 3.16: The *Xabsl2 Behavior Tester*, a part of the *RobotControl* application, makes use of the debugging interfaces of the *XabslEngine*.

Based on that interfaces, two debug tools were integrated into the *RobotControl* [61] application, the general debug tool of the *GermanTeam*. First, the *Xabsl Behavior Tester* (cf. fig. 3.16) allows to trace 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. Into the other direction, single options or basic behaviors can be selected and parameterized manually for execution.

Second, the *Xabsl Profiler* (cf. fig. 3.15) can be used to analyze behaviors over time. For that, log files containing the option activation path are recorded and visualized such that it can be seen for how long states and options were active. This helps to detect state oscillations or unused states.

3.6 Discussion

The *XABSL* system is a tool that can be used for decision making in autonomous agents. Because the language has no elements that are specific for a certain agent system and due to the independence of the runtime system *XabslEngine* from specific software platforms, *XABSL* can be applied in very different agent architectures and platforms.

That's why it depends on the chosen agent architecture and the implemented behaviors whether an *XABSL* agent behavior specification is reactive or deliberative. If the criterion for that distinction is that the environment is represented and modeled in persistent states, integrating past information, then it depends on if either the agent system directly passes the sensor readings to the *XABSL* behaviors or a world model is built up and made available. But as the state based approach tends to continue once selected behaviors, there are persistent states of intention. If seen from that perspective, *XABSL* is clearly deliberatively.

In the taxonomy of Russel and Norvig [66], *XABSL* agents are *goal based agents*, although there are no explicit goals. But implicitly the implemented behaviors (options) have goals, which are decomposed into sub-goals (subsequent options). Previous goals and intentions (option and state activations) are kept.

The architecture is hierarchical, as complex behaviors are composed from simpler ones. But it is not layered, because although more long-term and deliberative behaviors reside in higher levels of the hierarchy and more low-level and reactive behavior on lower levels, there is no conceptual differentiation between different levels of the option graph.

XABSL does not contain a classical planning component in the meaning that plans are derived automatically from the current world model or future simulations, but is possible to add such mechanisms to the agent system and to make the results available to the *XABSL* behaviors through input symbols.

The *XABSL* architecture is behavior based [7] as high-level behaviors are constructed from a set of reactive basic behaviors. Thereby is due to the use of finite state machines always only one basic behavior selected at the same time. But nevertheless, it is possible to combine different behaviors *continuously* [6] inside the basic behaviors, for instance by using potential fields.

The system is used inside of existing agent architectures for decision making. *XABSL* can neither be used to model a complete agent system nor is it able to control the complete agent program (instead, it is frequently called from the agent program). This is in contrast to many other languages such as the *Behavior Language* [14], *COLBERT* [37], or *CDL/ MissionLab* [48], which model complete agents including sensory and motor control capabilities.

Additionally and also in contrast to these systems, *XABSL* does not translate the behavior specifications into the code of the native programming languages (such as C++) but directly interprets an intermediate code. Thus, it is not necessary to recompile the programs if the behaviors change, leading to a shorter change-compile-test cycle.

The language can be best compared with the *Configuration Description Language* (*CDL*), a part of the *MissionLab* system. As CDL, *XABSL* allows to completely specify agent behavior based on hierarchies of finite state machines. But *XABSL* has a higher expressiveness in conditions for state transitions so that CDL documents could be transformed into *XABSL* documents, but not vice versa.

4 Applications

XABSL was initially developed for the *GermanTeam* (cf. sect. 4.1), a group of several German researchers competing in the *RoboCup* [36, 1, 8] *Sony Four Legged League*. As the first and main application, *XABSL* was used by them to model the overall behavior of soccer playing Aibo robots. Section 4.2 describes the implemented soccer behaviors of the *GermanTeam* in detail. Since 2004, it is also used to control the head movements of the robots (section 4.3).

Section 4.4 introduces an example *XABSL* behavior implementation for the *ASCII Soccer* environment [10]. This example was done to support behavior engineers when employing *XABSL* on their own agent platform. Additionally, it shows that the *XABSL* language, the tools and the executing engine are independent from the developments made for the Sony Four Legged League.

4.1 RoboCup and the GermanTeam

The *GermanTeam* [2] competes in the Sony Four Legged League and is a cooperation of four German RoboCup teams: *Aibo Team Humboldt* (Humboldt-Universität zu Berlin), *Bremen Byters* (Universität Bremen), *Darmstadt Dribbling Dackels* (Technische Universität Darmstadt), and *Microsoft Hellhounds* (Universität Dortmund). The *GermanTeam* is a national team. The members participated as separate teams in the national German Open competitions in Paderborn 2001, 2002, 2003, and 2004 but formed a single team at the international RoboCup world championships in Seattle 2001, Fukuoka 2002, Padova 2003, and Lisbon 2004. The author is member of the *Aibo Team Humboldt* from the Humboldt-Universität zu Berlin.

4.1.1 The Sony Four Legged League

The Sony Four Legged League is one of the official leagues in RoboCup. The robot platform in this league is standardized. The Sony Aibo [25] ERS-210, ERS-210A (cf. fig. 4.1a), and ERS7 (cf. fig. 4.1b) are the only permitted systems in 2004, and can only be used without hardware modification. The main sensor of the Sony Aibo is the camera located in its head. The camera serves 25 color images per second with a resolution

a) b)

Figure 4.1: a) The Sony Aibo ERS210A, with red tricot. b) The Sony Aibo ERS7, without team markers.

of 176×144 pixels (ERS7: 30 fps, 208×160 pixels). The robots are equipped with a single 200 MHz MIPS processor (ERS-210A: 400 MHz, ERS7: 576 MHz) and 32 MB of RAM (ERS7: 64 MB). Moreover, the robot has touch sensors in the back and the head, three acceleration sensors, two microphones in the ears, and an infrared distance sensor in the head (ERS7: additional infrared sensor in the chest). A WLAN card allows to communicate with team mates or a PC for debugging purposes.

The head has three degrees of freedom and each of the four legs has three joints. The tail, the mouth and the ears can also be moved. LEDs in the head and the tail (ERS7: on the back) and a speaker in the mouth allow additional visual and acoustic output.

The soccer field in the Sony Four Legged League approximately has a size of 5m×3m (cf. fig. 4.2). As the main sensor of the robot is a camera, all objects on the RoboCup field are color coded. There are four two-colored flags for localization in the corners (pink and either yellow or sky-blue), the two goals are of different color (yellow and sky-blue), the ball is orange (as in all RoboCup leagues), and the robots of the two teams wear tricots in different colors (red and blue).

A soccer game lasts 2x10 minutes. The robots act completely autonomous, i.e. there is no external computer beside the field that can help the players in their calculations. The only exception is the so-called *game manager* operated by the referee which allows to remotely start and stop a game.

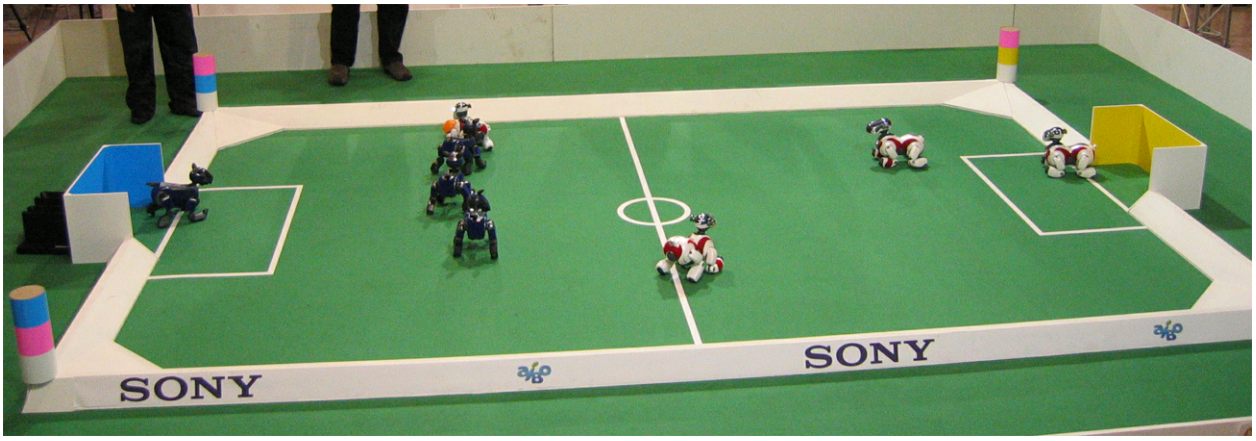


Figure 4.2: The soccer field in the Sony Four Legged League. On the left a team of four Sony Aibo ERS210 robots, on the right four Sony Aibo ERS7 robots.

4.1.2 Characteristics of the Sony Four Legged League

The Sony Four Legged League (as well as the Humanoid League) differs from the "wheel based leagues" in the complexity of physical actions that have to be employed both for interaction and perception. Instead of kicking with a single kicking device such as in the middle or small sized league, this allows for a lot of different kicking skills using legs, body, or even head, which sometimes require preparatory movements. Instead of moving on wheels many different styles of omni-directional walking are used in different situations.

As the camera of the robot has only a very narrow opening angle, the problem of directed vision (in contrast to omni-vision as often used in the middle-sized league or in the first approaches of the small-sized league to local vision systems) has to be tackled. And as information is collected and modeled over time, the movement of legs and head has to be coordinated with current vision needs.

Although the quality and reliability of the perception and world modeling capabilities of the *GermanTeam*'s robots constantly increases [34, 33, 32, 29, 28, 19, 62, 63, 64], the resulting world model is still very uncertain and incomplete, which is one of the most challenging problems for behavior control in the Sony Four Legged League. The results of the performed actions are also very unpredictable. A kicked ball, for example, rarely rolls into the desired direction. Until now, the robots are only poorly able to recognize whether they are stuck to other robots.

With the introduction of Wireless LAN communication in the Sony League in 2002, cooperative strategies became more complex and consequently require adequately formulated high level behaviors. The problem to be faced here is the low bandwidth and the sometimes relatively long transmission times for messages, which makes team coordination a difficult task.

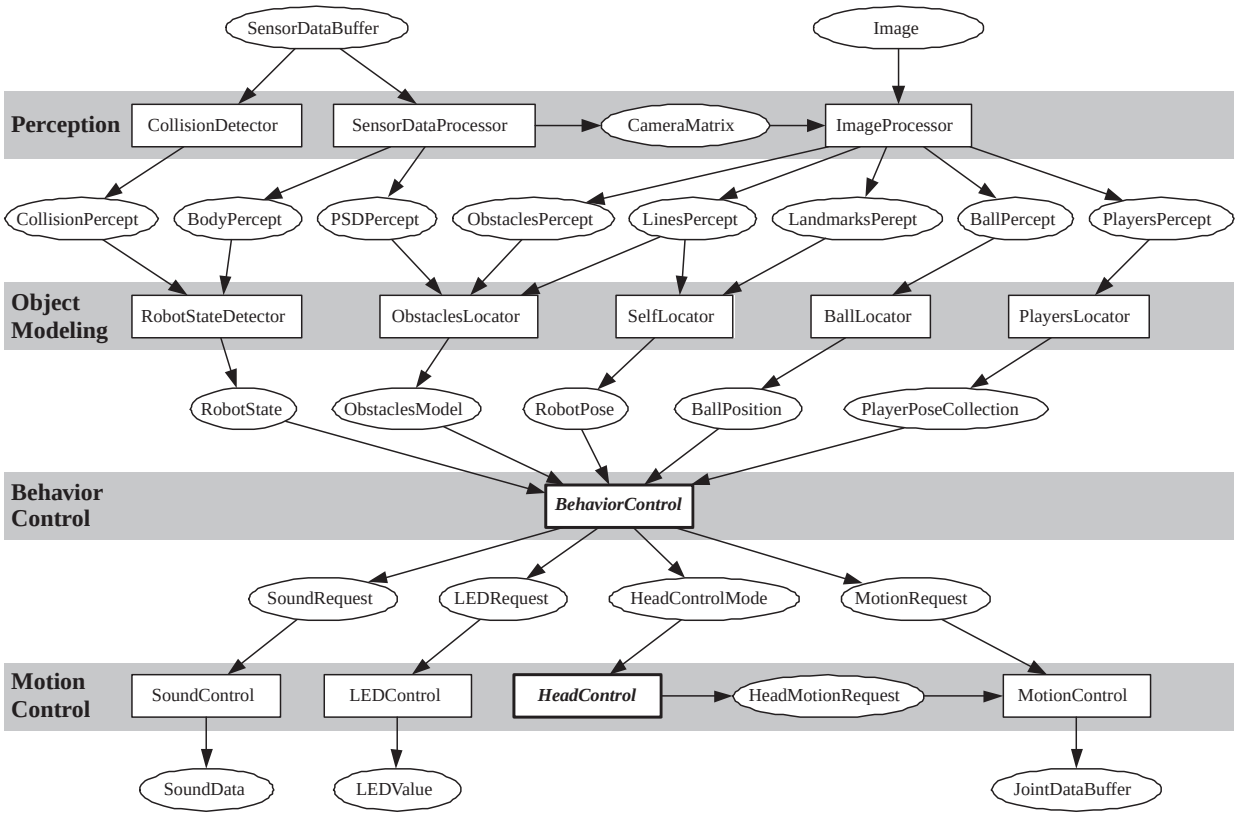


Figure 4.3: Information processing in the robots of the GermanTeam. Boxes denote modules, ellipses denote the representations that are needed to exchange information between the modules. The bold boxes mark the both modules which are based on *XABSL*: *"BehaviorControl"* and *"HeadControl"*.

4.1.3 The Software Architecture of the GermanTeam and XABSL

The *GermanTeam* divided the information processing of the robots into a set of well-defined tasks such as image processing, ball modeling, or LED control [61]. Each of these tasks is performed by a *module*, a "sub program" with well-defined interfaces that allow to develop multiple switchable *solutions* for a task. The modules exchange information only via external representations – the interface of a module defines, which representations are the input of the module and which representations have to be written as output. The modules can be distributed freely over different concurrent processes.

A simplified graph of the *GermanTeam's* modules is shown in figure 4.3. On the first level, the perception modules process the raw sensor data and generate percepts, e.g. the *"BallPercept"* or the *"LandmarksPercept"*. The modules on the second level try to integrate these percepts over the time into a stable world model. The third level uses the items of this world model such as *"BallPosition"* or *"RobotPose"* to select appropriate actions and to generate commands for the actuator control modules on the fourth level.

In the current process layout of the *GermanTeam*, the modules from the first three levels are executed together in the process *"Cognition"*, which is triggered every 33 ms by a new image from the robot's camera. Concurrent to that, the modules of the fourth level are executed together every 8 ms in the process *"Motion"*. The both modules which use *XABSL* are *"BehaviorControl"* on the third and *"HeadControl"* on the fourth level.

The *"BehaviorControl"* module has no access to the data gained at the perception level but makes its decisions based on the information provided by the world modeling modules. The main items of this world model are the robot's pose, the ball's position and speed modeled from own observations, the ball position communicated by team mates, the positions of the team mates, an vision based obstacle model which contains 90 sectors with distances to the next obstacles, a robot state that contains information about the switches of the robot and whether the robot has fallen down, the game control data received from the *RoboCup Game Manager* application, and behavior messages from team mates. Additionally, the motion control modules provide information about the currently executed motions, as there is a delay between the request of a motion and the start of a motion.

The primary task of the module *"BehaviorControl"* is to control the leg movements of the robot by generating a motion request that is executed by the motor control modules. This is either a request to walk with a specific walk type, translation speed, and rotation speed or a request for a special action such as a kick, getup, or stand. Although the motion system is responsible for the control of the head movements, the behavior control programs have to set a head control mode that specifies preferences where to direct the head to. A LED request specifies how to control the LEDs of the robot and a sound request allows to play wave files for debugging purposes. Finally, the behaviors can send messages to the team mates.

The second *XABSL*-powered module, *"HeadControl"* has to control the three joints of the head by writing a head motion request. It carries out the head control mode which was set by the behavior control. To be able to direct the camera to the ball and other interesting objects on the field, it has access to the robot's pose, the ball's position and speed, a body posture determining the tilt and the height of the robot's body, and information about the currently executed leg motions.

4.1.4 History of Development

The *XABSL* system evolved together with the programs of the *GermanTeam* over three years. A C++ implemented layered state machine architecture was developed for the RoboCup competitions in 2001 in Seattle [15]. Based on that, a first version of *XABSL* was developed for the participation of the *Aibo Team Humboldt* [3] at the German Open 2002 in Paderborn. Later, this approach was chosen for the participation of the *GermanTeam* in the RoboCup competitions 2002 in Fukuoka (Japan) [20]. In 2003, the language, the tools and the behaviors themselves were largely improved and used by all *GermanTeam* members for the German Open 2003. The resulting four different solutions could be easily merged into a common behavior solution for the participation of the *GermanTeam* in the RoboCup world championship 2003 in Padova (Italy) [62]. In 2004, again all members of the *GermanTeam* used the system for their participation in the German Open 2004. As almost no changes were made to *XABSL* itself, the main focus of development was in the behavior implementations itself, resulting in the win of the RoboCup world championship 2004 in Lisbon.

4.1.5 Developing Agent Behaviors in a Team

More than 20 team members of the *GermanTeam* were involved in the developing and tuning of the behaviors of the *GermanTeam*. The modular approach of *XABSL* supports the development of behaviors in a team. Many developers can easily extend or advance the behaviors in parallel. The distribution of an *XABSL* agent behavior specification over many files simplifies the use of a version control system such as CVS.

New options can easily be added to existing ones without having negative side effects. The debugging interfaces of the *XabslEngine* allow it to test options separately before they are used by more high-level options. Better solutions of existing options can be developed in parallel and are easy to compare with the previous ones. A constantly increasing library of well tuned low level behaviors can be reused in different contexts for the creation of new options.

All four member universities of the *GermanTeam* used a separate *XABSL* system for their participation in the German Open 2003 and 2004. Nevertheless, due to the modular constitution of *XABSL* behavior specifications, it was an easy task to merge afterwards the best behaviors (options) of the four universities into a common solution for the participation in the RoboCup world championships 2003 and 2004.

4.2 Playing Soccer with XABSL

The behaviors which the *GermanTeam* developed in *XABSL* for the RoboCup championships 2004 in Lisbon are distributed among about 60 options. Figure 4.4 shows the option graph of the soccer related behaviors.

In general, the lower behaviors in the option hierarchy such as ball handling or navigation, have to react instantly on changes in the environment and are therefore very short-term and reactive. The more high-level behaviors such as waiting for a pass, positioning, or role changes try to prevent frequent state changes to avoid oscillations and make more deliberative and long-term decisions. This section describes from bottom to top how the *GermanTeam*'s robots play soccer starting with basic capabilities and finishing with the high-level team strategies.

An extensive automatically generated HTML documentation of these behaviors can be found at *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2004/*. It is recommended to use this site as an additional source to this Section.

4.2.1 Ball Handling

The *GermanTeam* won the 2004 RoboCup world championships due to – besides other things – its sophisticated well tuned ball handling behaviors. They are composed from 18 options and 7 basic behaviors, which looks much. But this section will show how step by step the whole behavior is composed from simple options in a clear and straight forward way.

4.2.1.1 Approaching

All behaviors for ball approaching and dribbling are based on one single basic behavior: *"go-to-ball"* is responsible for walking to the ball. For the use in different contexts, it provides a variety of parameters. First, the body of the robot is always directed to the ball, restricted by the parameter *"max-turn-speed"*. The maximum speed is given by the parameter *"max-speed"*, making higher options responsible for slowing down near the ball. The *"max-speed.y"* parameter restricts the sideward component, allowing for sprinting with the "dash" walk type. For dribbling and the "turn kick" (cf. 4.2.1.2), *"y-offset"* specifies a y offset with that the robot shall arrive at the ball. If the robot is very close to the ball and if the ball is very to the left or right, the translation component is almost completely inhibited, making the robot only turn in order to avoid pushing the ball away with the front legs.

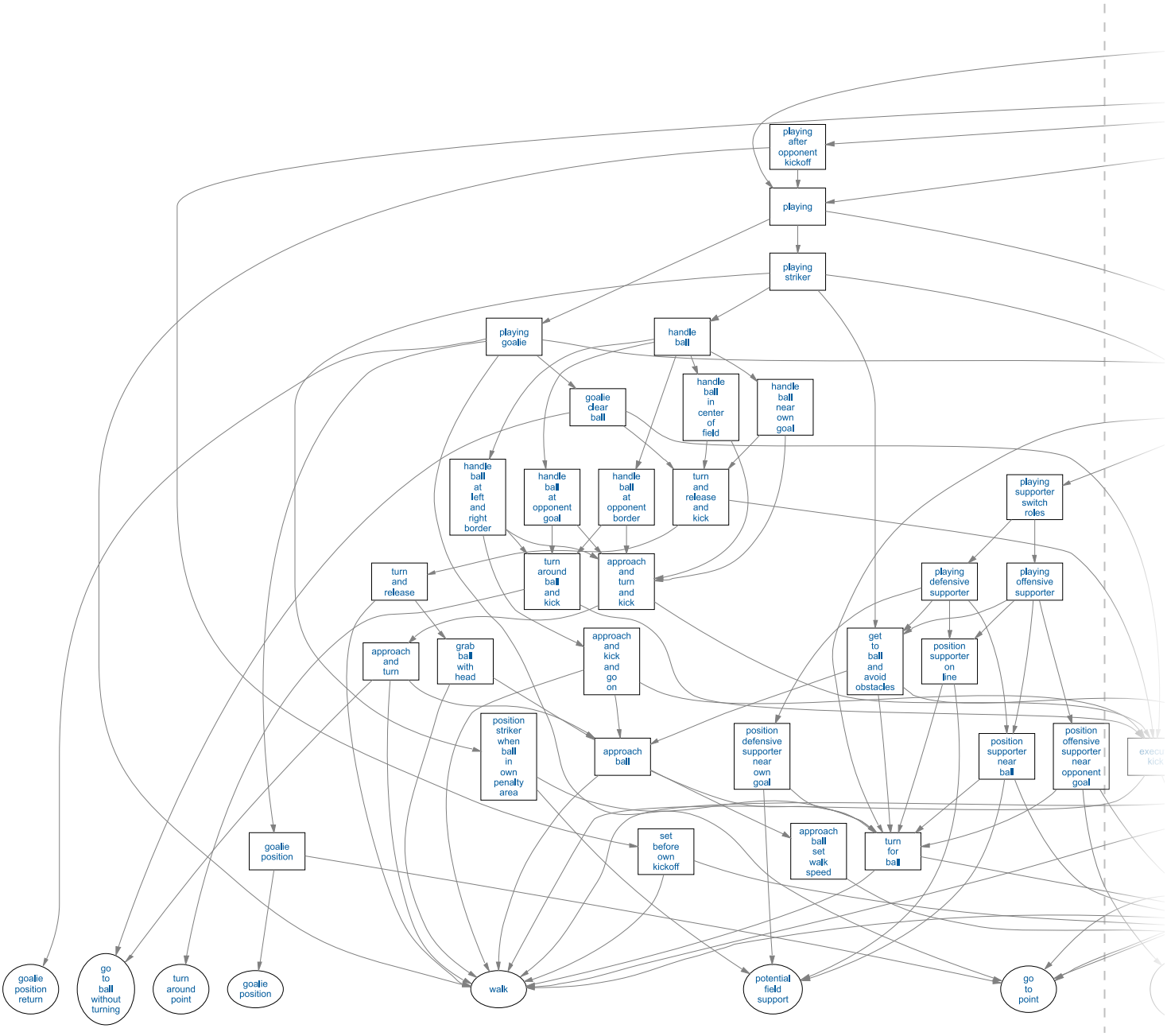


Figure 4.4: The option graph of the soccer-related behaviors of the *GermanTeam*.

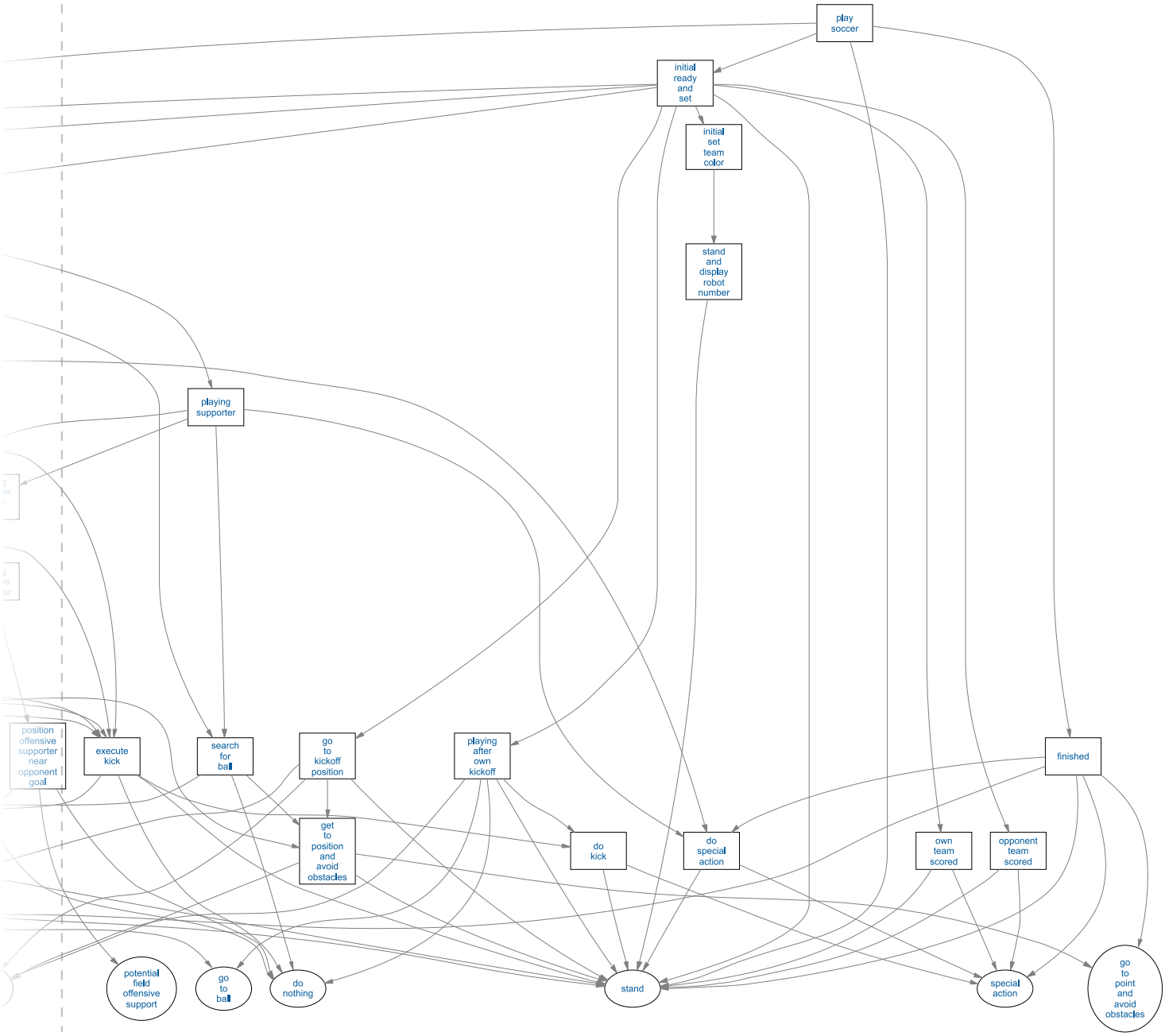

play soccer
initial ready and set
initial set team color
stand and display robot number
playing supporter
position offensive supporter near opponent goal
execute kick
search for ball
go to kickoff position
playing after own kickoff
get to position and avoid obstacles
do kick
do special action
own team scored
opponent team scored
finished
potential field offensive support
go to ball
do nothing
stand
special action
go to point and avoid obstacles

Figure 4.5:

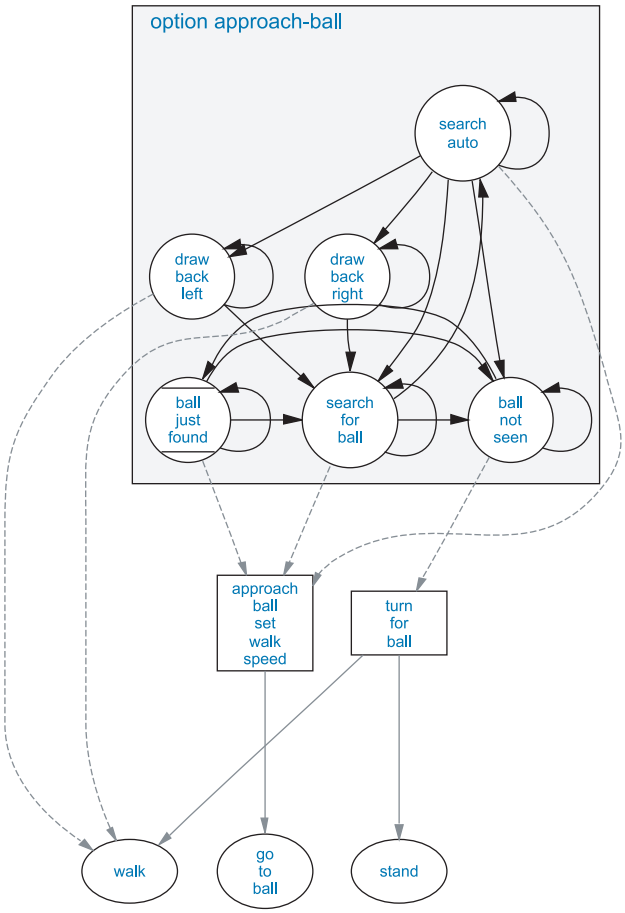


Figure 4.6: Option *"approach-ball"* controls the head movements while approaching the ball and handles collisions and ball losses. The complete documentation of the option can be found at *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2004/option.approach-ball.html*.

The ball handling behaviors do not reference the *"go-to-ball"* basic behavior directly but use the option *"approach-ball"* (cf. fig. 4.6). This option makes a distinction whether the robot is far away from the ball or close. In the first case, in state *"search-auto"*, the head-control mode *"search-auto"* is set. This lets the head of the robot look at the ball and – frequently, always after a certain time of consecutively perceived balls – shortly look around for landmarks and obstacles to improve self-localization. These head scans are disadvantageous near the ball. That's why if the robot gets closer to the ball than specified in the option parameter *"look-at-ball-distance"*, in state *"search-for-ball"* the head control mode is set to *"search-for-ball"*. This lets the head only look at the ball. To avoid frequent changes between these two states, there is a distance hysteresis of 5 cm between them. In both states, the option *"approach-ball-set-walk-speed"*, which controls the walk speed (see below), is referenced.

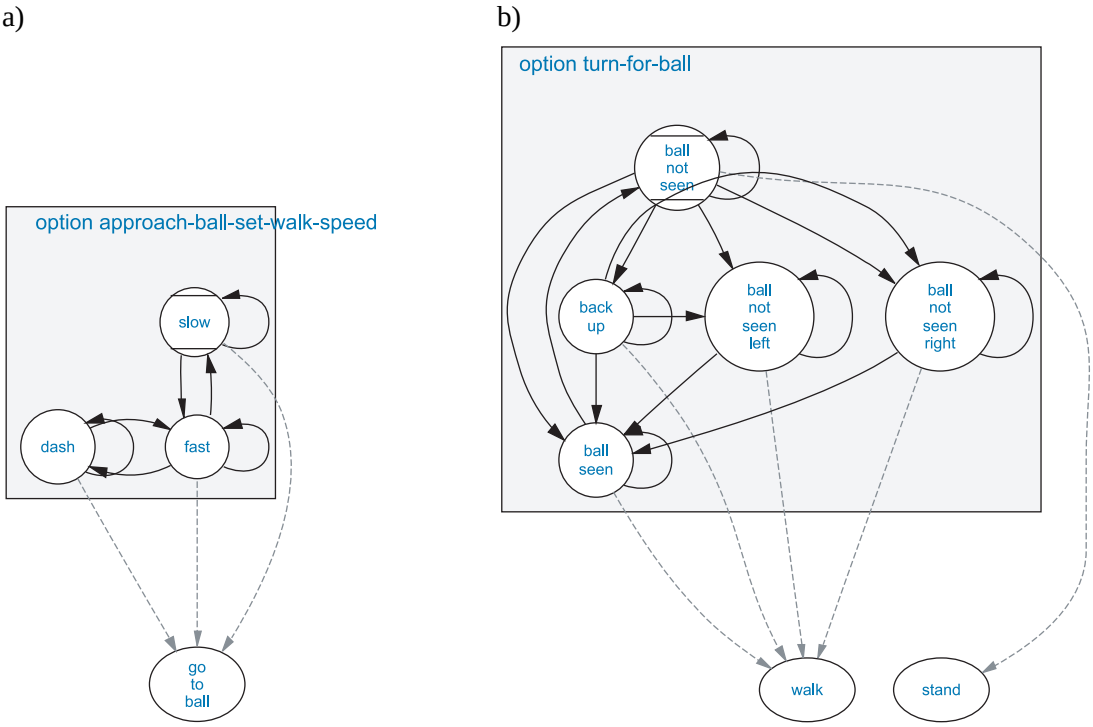


Figure 4.7: a) Option *"approach-ball-set-walk-speed"* controls the speed of ball approaching. b) Option *"turn-for-ball"* tries to redetect a previously lost ball.

If the robot is far away from the ball (in state *"search-auto"*) and there is a collision with another robot (detected as described in [28]), in the states *"draw-back-left"* and *"draw-back-right"* the robot walks sideways for a short time to uncouple from the other robot. There is no transition from *"search-for-ball"* to the draw back states in order to give opponent robots no advantage near the ball.

If the ball was not seen for 1.3 seconds in the *"search-auto"* state or not for 400 ms in the *"search-for-ball"* state, in state *"ball-not-seen"* the option *"turn-for-ball"* (see below) tries to redetect the ball. If the ball is seen again, the state *"ball-just-found"* remains active for 2 seconds, setting the head control mode *"search-for-ball"* in order to avoid further ball losses due to scanning around with *"search-auto"*.

Option *"approach-ball-set-walk-speed"* (cf. fig. 4.7a) controls the speed of ball approaching. It is only used by option *"approach-ball"*. In state *"fast"*, the basic behavior *"go-to-ball"* is executed with a fixed speed of 350 cm per second. If the robot gets closer to the ball than specified in parameter *"slow-down-distance"* (minus 2,5 cm hysteresis), in state *"slow"* the speed given in *"slow-speed"* is passed to *"go-to-ball"*. From the *"fast"* state, if the ball is farer away than 1200 cm and if the angle to the ball is

between plus and minus 7 degrees, state *"dash"* becomes active. There *"go-to-ball"* is executed with walk type *"dash"*, a faster but not omnidirectional walking gait.

The ball approaching stops immediately after the ball was not seen anymore for a certain time (see above). In this case, *"approach-ball"* references the option *"turn-for-ball"* (cf. fig. 4.7b) to redetect the ball. In the initial state *"ball-not-seen"*, the basic behavior *"stand"* is executed. Note that *"stand"* does not stop walking immediately but continuously slows down in order to avoid bumpy movements if the ball is redetected fast. As *"turn-for-ball"* can be activated from different contexts and situations, the time how long state *"ball-not-seen"* remains active depends on how long the ball was not seen and where it was seen last. The state remains active for at least 800 ms which are needed for *"stand"* to almost slow down. As long as the ball was seen 1.7 seconds before, *"ball-not-seen"* keeps active to give the head control a chance to make a complete scan around. If the ball was seen in the last 5 seconds and in the near, it is very likely that the ball is at the side of the robot. Therefore, in state *"back-up"* the robot walks backward for 1.5 seconds to redetect the ball. If that fails (or from *"ball-not-seen"* if all other conditions fail), the state *"ball-not-seen-left"* or *"ball-not-seen-right"* gets active, depending on whether the ball was previously seen left or right. The robot turns around using the *"walk"* basic behavior. The head control mode is set to *"look-left"* or *"look-right"*, letting the robot look into the direction of turning. Although the *"turn-for-ball"* option is not activated anymore from *"approach-ball"* when the ball is redetected, state *"ball-seen"* becomes active when the ball is seen again, turning the robot to the ball.

4.2.1.2 Dribbling

The option *"approach-and-turn"* (cf. fig. 4.8) dribbles the ball into the direction that is passed through the parameter *"angle"*. Already this behavior is able to get the ball reliably into the opponent goal. It is composed from mainly *"approach-ball"* and a few other short walk sequences that push the ball into the desired direction. It does not use any kicks which makes it a fast and robust behavior.

State *"approach-ball"* activates option *"approach-ball"* with proper parameters for fast ball approaching. When it happens that the ball is very close and that the robot is already directed into the direction where the ball shall be dribbled to, state *"go-to-ball-without-turning"* is activated and basic behavior *"go-to-ball-without-turning"* is selected. Different from *"go-to-ball"*, this basic behavior uses only x and y translation. The advantage is that it is a bit faster near the ball than the normal ball approaching. As soon as the conditions above are not met anymore (with a hysteresis), the option returns to the state *"approach-ball"*.

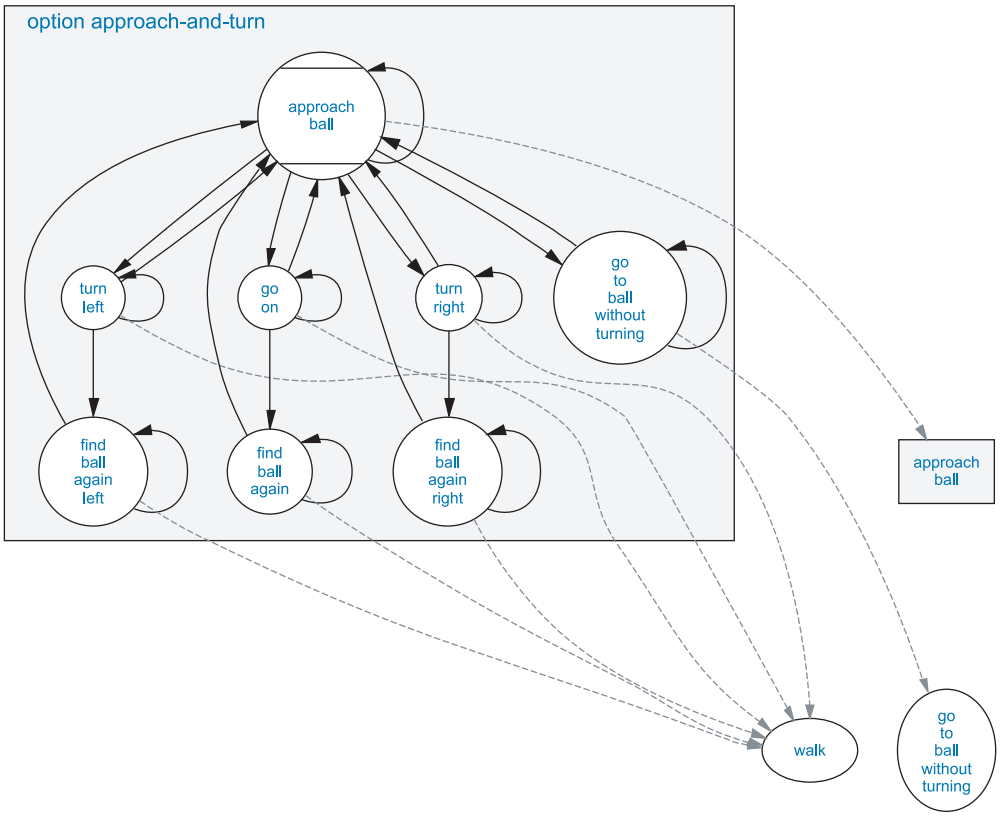


Figure 4.8: Option *"approach-and-turn"* dribbles the ball into given direction by pushing the ball with the chest or pulling it around with the front legs.

If the ball is seen well and directly in front of the robot, one of the three dribbling moves starts: state *"turn-right"* becomes activated if the destination is more to the right than 30 degrees, *"turn-left"* gets active if the destination is more to the left than -30 degrees, and otherwise state *"go-on"* is chosen. In *"go-on"*, the robot just runs blindly straight ahead for 250 ms, pushing the ball forward with the front legs or chest. If after the time the ball is seen again or still seen, it is returned to state *"approach-ball"*. Otherwise, in state *"find-ball-again"*, the robot does not stop – as it would happen when in the *"approach-ball"* option the ball is not seen anymore – but still walks forward with a slow speed for maximum 500 ms, assuming that the ball is still in front of the robot and not at the side or behind.

In the states *"turn-left"* and *"turn-right"*, the robot simultaneously walks forward and turns at the same time for 500 ms, pushing the ball reliably and strong into a direction of approximately 60 degrees. A different walk type, *"turn-kick"* is used in order to have the front legs more stretched to the front for safer guiding the ball with the outer leg. If the ball is not seen after the 500 ms, in the states *"find-ball-again-left"* and *"find-ball-again-right"* the robot does not stop but also walks straight ahead at a slow speed. Additionally,

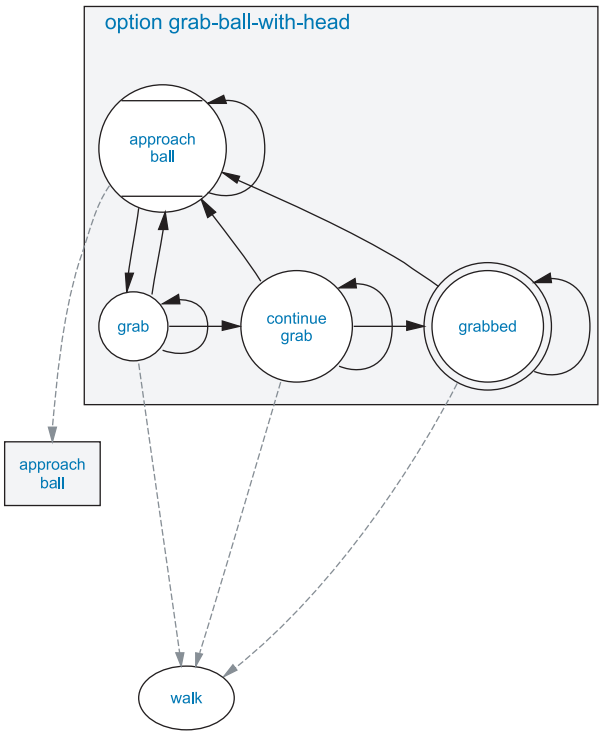


Figure 4.9: Option *"grab-ball-with-head"* grabs the ball with the head.

the head control mode *"search-for-ball-left"* or *"search-for-ball-right"* is set, which gives the head control a hint in which direction the ball was pushed and where to search first.

4.2.1.3 Grabbing and Pushing Backward

The dribbling with *"approach-and-turn"* is only reasonable when the robot is behind the ball (seen from the direction where the ball shall be played to). For all other cases, option *"turn-and-release"* grabs the ball with the head (the ball is shut between the front legs and the head), turns with the grabbed ball, and then releases the ball again.

The behavior for ball grabbing is encapsulated in a separate option, *"grab-ball-with-head* (cf. fig. 4.9). In the initial state *"approach-ball"*, option *"approach-ball"* is selected with a quite low speed near the ball. If the ball is in the correct position for grabbing, in state *"grab"* the robots walks forward at a low speed "onto the ball". The head control mode is set to *"catch-ball"* and the actual job of grabbing is done by the head control. The infrared distance sensor in the chest is used to measure the exact distance to the ball. If the ball is not at the chest yet, the head is lifted in order to push the ball not away with the head. Otherwise, the head is bended over the ball. The state *"grab"* is active without feedback for 1 second. After that, in state

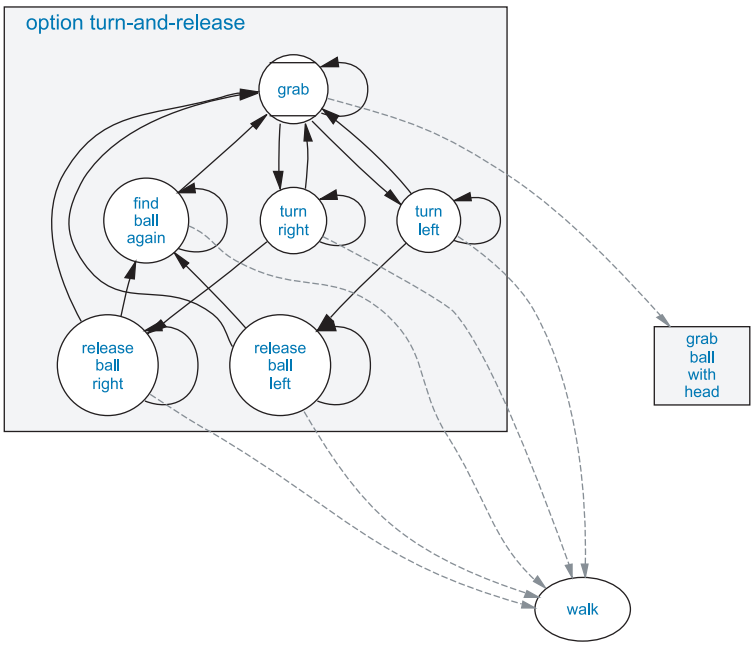


Figure 4.10: Option *"turn-and-release"* grabs the ball and pulls it around. The actual job of lifting the head in the right moment is done in the head control.

"continue-grab" it is checked with the infrared distance sensor whether the grab was successful (it has to be checked both in the head control and in the behavior control as a transmission of this information from the *Motion* to the *Cognition* process would last too long). If not, it starts from the beginning in the *"grab"* state. If the ball was grabbed, the option stays in the state *"grabbed"*, which is a target state to signal higher options that the whole behavior was successful.

In the initial state *"grab"* of option *"turn-and-release"* (cf. fig. 4.10), the option *"grab-ball-with-head"* is executed until it reaches its target state. After that, in the states *"turn-left"* and *"turn-right"*, the robot turns with the ball to the desired direction given by the option parameter *"angle"*. The head control mode is set to *"catch-ball"* in order to keep the ball grabbed. The special walk type *"turn-with-ball"* is set. With that, the robot uses almost only the hind legs for turning, the front legs enclosing the ball. After the difference to the target angle gets less than 110 degrees, in the states *"release-ball-left"* and *"release-ball-right"* the ball is released again. The head control mode is set to *"release-caught-ball-when-turning-right"* and *"release-caught-ball-when-turning-left"*. While the robot just continues to turn with walk type *"turn-with-ball"*, again the actual job is done by the head control. To give the ball a strong push with the outer leg, the head is lifted only if the current position in the walk cycle is between 0.77 and 0.85 when turning right

or between 0.27 and 0.35 when turning left. If state and option *"approach-ball"* would be activated direct after that, the ball would be assumed to be lost as it indeed was not seen during turning. Therefore, in state *"find-ball-again"*, the robot has a chance to redetect the ball while slowly walking forward for maximum 500 ms.

4.2.1.4 Kicking

Kicking fast and precisely is crucial when playing robot soccer. Thus, the *GermanTeam* developed about 50 different kicks, suitable for almost all situations that can happen during a match. This large amount of specialized kicks requires a proper evaluation method to select which kick should be used in a certain situation.

Thereto, the *GermanTeam* followed three main goals for kicking: First, the robots should be able to play without any kicks. This goal was achieved first by implementing options like the above discussed *"approach-and-turn"* and *"turn-and-release"*. Second, there should be no actions that try to establish a special situation in that a kick can be applied (like strafing or exact positioning at the ball). Instead, the robots should play the ball as if they were not using any kicks and kicks should be performed only if there was by chance an appropriate situation. And third, the kick selection itself should be more flexible, easy to extend, and, above all, not in *XABSL*, as it is indeed possible but very hard and time consuming to model and fine-tune the prerequisites of a kick in *XABSL*.

The goals two and three were achieved by introducing *kick selection tables*. A kick is retrieved from such a table by putting in the desired kick direction and the current x and y position of the ball. The look-up table stores for 12 discrete sectors (30 degrees each) of desired kick directions the start positions of appropriate kicks in a 1 cm wide grid, as shown in figure 4.11a) and c). To gain the table, a semi-autonomous teach-in mechanism was developed. Thereto, a robot stands on the playing field and kicks the ball several times with the same kick. Meanwhile, the starting position and the final position of the ball are measured relative to the robot. The results of such kick experiments can be seen in figure 4.11b) and d). For editing the kick selection table based on that data, a kick editor was developed.

As different situations on the field require different kicks, there are multiple kick selection tables. There are ones for the goalie, a field player playing in the center of the field, near the own goal, at the right border, at the left border, at the left opponent border, at the right opponent border, near the own goal, and near the opponent goal (cf. sect. 4.2.1.5).

To make the data stored in the kick selection table accessible to the *XABSL* behaviors, *XABSL* constants

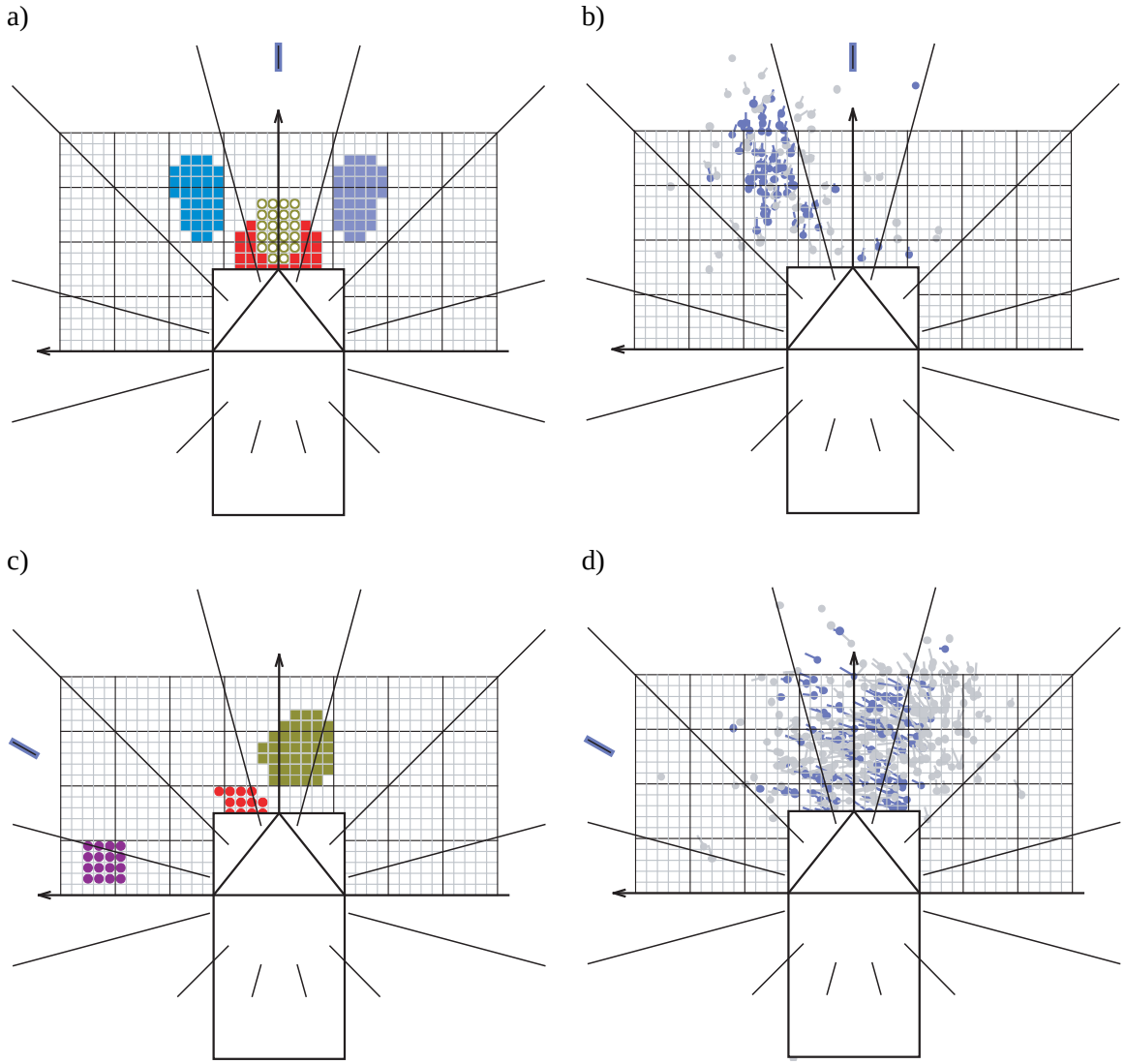


Figure 4.11: a) The kick selection table for the goalie when the desired kick direction is “forward” (in the sector between -15 and 15 degrees). If the current position of the ball is in the outer blue areas, the *“left-paw”* or *“right-paw”* kick is selected, in front of the robot (red area), kick *“chest-strong”*, and in a narrow range more distant in front of the robot (brown area) *“forward-kick-hard”*. b) Data recorded from kick experiments for the *“left-paw”* kick. The dots mark the position where the ball was perceived before the kick started. The lines out of the dots indicate in which direction and how far the ball was kicked in the experiment. All kick experiments in that the ball was kicked into the “forward” sector are highlighted blue. The area for *“put-left”* in a) was defined by taking these highlighted entries into account. c) The goalie kick selection table for the sector between 45 and 73.5 degrees. For the ball to the very left (purple area), *“put-left”* is selected, close to the robot (red area) *“hook-left”*, and in the brown area *“head-left”*. d) Kick experiments for the *“head-left”* kick, with those entries highlighted where the ball was kicked into the direction between 45 and 75 degrees.

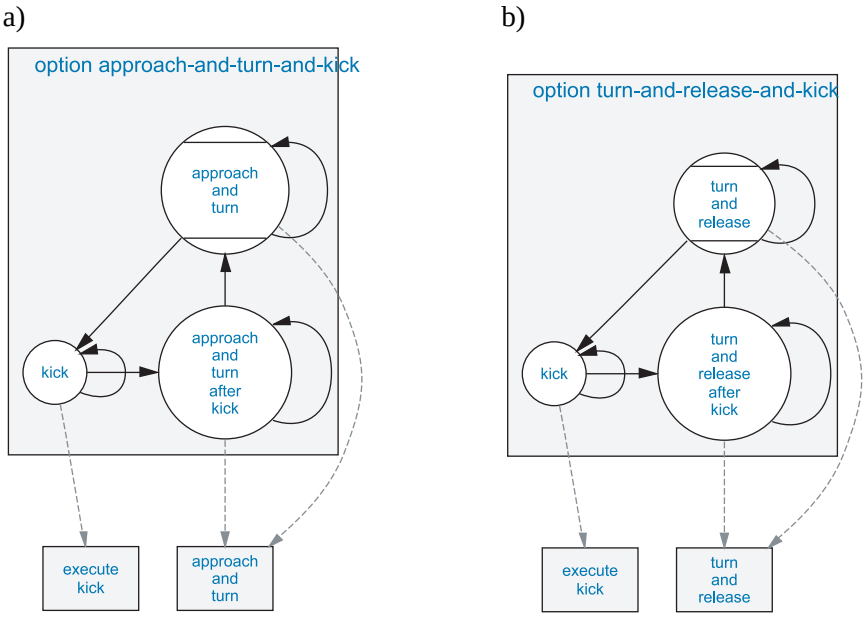


Figure 4.12: Both a) option *"approach-and-turn-and-kick"* and b) option *"turn-and-release-and-kick"* execute a behavior that is able to play the ball without kicking and only perform a kick if by chance it is applicable.

for the table and kick ids are generated automatically from the C++ implemented kick selection table. The decimal input function *"retrieve-kick"* is used to retrieve a kick, taking the desired kick direction and the id of the table to be used as parameters. If the returned kick is different from *"action.nothing"*, an appropriate kick was found for the current situation and the kick can be executed by using the option *"execute-kick"*.

Option *"approach-and-turn-and-kick"* (cf. fig. 4.12a) is an example for a behavior that uses kicks. It is composed from a behavior that is able to play the ball without kicking (*"approach-and-turn"*) and the kick execution option *"execute-kick"*. In the initial state *"approach-and-turn"*, the kick selection table is always queried whether a kick is possible. If so, the kick is executed in the state *kick*. After it finished (the option *"execute-kick"* reached its target state), option *"approach-and-turn-and-kick"* remains for 2.5 seconds in the state *"approach-and-turn-after-kick"*, which executes the same behavior as state *"approach-and-turn"* but makes sure that there elapse at least 2.5 seconds between two successive kicks.

Similarly, the option *"turn-and-release-and-kick"* (cf. fig. 4.12b) is composed of *"turn-and-release"* and *"execute-kick"*. In the option *"turn-around-ball-and-kick"*, the basic behavior *"turn-around-ball"* turns the robot behind the ball, which is needed at the borders of the field (*"turn-and-release"* does not work there). Option *"approach-and-kick-and-go-on"* uses only *"approach-ball"* but has an additional state *"go-on"*,

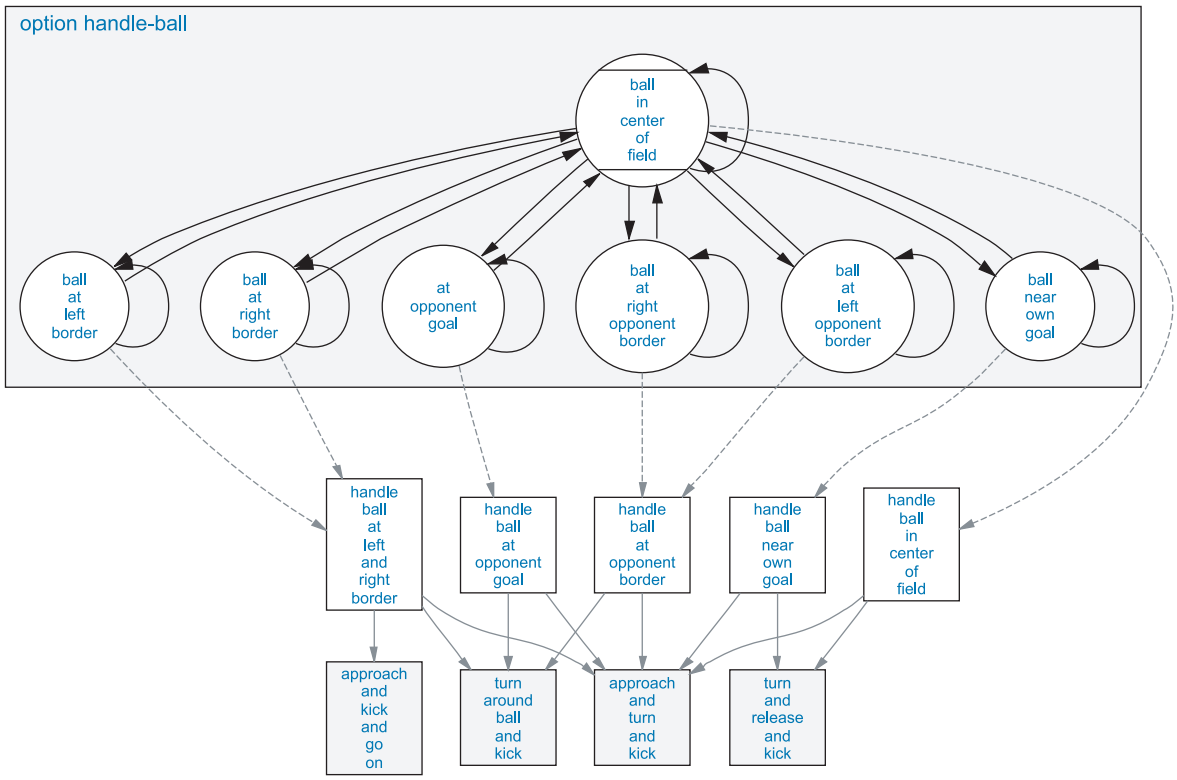


Figure 4.13: Option *"handle-ball"* selects between different behaviors for different zones on the field.

similar to in *"approach-and-turn"*.

4.2.1.5 Zones for Ball Handling

Some zones of the field require different behaviors than when playing in the center of the field. For instance, if the ball is at one of the borders, it is often not possible to grab the ball with the head. Instead, if the robot is in front of the ball, it has to turn behind the ball before it can be dribbled or kicked. Near the own goal, the direction where to play the ball is not that important as to clear the ball just somewhere. And, at the opponent goal, there is much more precision needed than in the rest of the field.

Option *"handle-ball'* (cf. fig. 4.13) selects between these behaviors depending on where the ball is on the field. The initial state *"ball-in-center-of-field"* covers most of the area of the field, executing option *"handle-ball-in-center-of-field"*. At the borders and near the goals there are separate states, executing the corresponding ball handling options. To avoid oscillations between these states, there was added a broad distance hysteresis. For instance, there is a transition from *"ball-in-center-of-field"* to *"ball-at-left-border"* when the y position of the ball is greater than 1250 mm. If then the y position of the ball gets less than 1100

mm, there is a back transition to *"ball-in-center-of-field"*.

In the center of the field, option *"handle-ball-in-center-of-field"* selects only between *"turn-and-release-and-kick"* and *"approach-and-turn-and-kick"*, depending on whether the robot is behind the ball or not. As the desired direction of play the *"best-angle-to-opponent-goal"* is passed. This angle is mostly the direct angle to the opponent goal. If there are obstacles on the way there, the angle is bended to the bigger gap in the obstacles. If there are everywhere obstacles in the direction of the goal, the angle to the next team mate is chosen, which sometimes results in a pass.

Near the own goal, option *"handle-ball-in-center-of-field"* uses the same options as in the center of the field, but passes a different angle: the *"best-angle-away-from-own-goal"* is not directed to the opponent goal but away from the own goal.

At the opponent goal, option *"handle-ball-at-opponent-goal"* combines *"approach-and-turn-and-kick"* with *"turn-around-and-ball-and-kick"*, using the angle *"angle-to-point-behind-opponent-goal"*. The turning behind the ball is indeed slower than when doing a kick to the side, but it is safer when opponent players such as the goalie are involved.

At the left and right border, option *"handle-ball-at-left-and-right-border"* chooses between *"approach-and-turn-and-kick-and-go-on"* if the robot is completely behind the ball, *"approach-and-turn-and-kick"* if the robot is almost behind the ball, and *"turn-around-ball-and-kick"* if not. The average distance to the ball over two seconds is used to decide whether the robot got stuck to other robots. If so, the kick selection table *"when-stuck"* containing quite imprecise but strong kicks is used. If not, less kicks are performed.

Option *"handle-ball-at-opponent-border"* has a very similar structure but uses less aggressive kicks to dribble the ball securely along the border into the opponent goal.

4.2.1.6 Transitions Between Ball Handling Behaviors

In the more high-level options, it is important to take into account when to do transitions between different behaviors. In general, all ball handling behaviors should be such that it is no problem to switch between them (which does not allow for strafing behaviors or behaviors for exact positioning for a certain kick). But there are some phases in behaviors such as ball grabbing, dribbling, or kicking, in that the behaviors should not be interrupted.

In *XABSL*, options have no chance to determine whether an option deep below in the option graph is in such a critical state. Therefore, the information whether the ball is handled at the moment is transmitted through an external variable, which can be queried through the Boolean input symbol *"ball.is-handled-at-*

the-moment". In the dribbling and kicking options, all states that execute a behavior that should not be interrupted set this variable by setting the enumerated output symbol *"ball.handling"* to *"handling-the-ball"*. In options higher in the option hierarchy, there are only transitions between states if *"ball.is-handled-at-the-moment"* is false.

Another principle for gaining smooth ball handling performance is that the higher the behavior in the option hierarchy, the less frequent should be transitions between states. A once selected behavior should be always continued unless there is a strong reason to change it.

Furthermore, if it happens that the ball is not seen anymore, the previously executed behavior should be continued until the ball is redetected (all ball handling options are based on *"approach-ball"*, which autonomously tries to redetect the ball using the *"turn-for-ball"* option). Therefore, there are only transitions in the higher options when the ball is just seen.

4.2.2 Navigation and Obstacle Avoidance

Navigation includes fast walking to a position with and without obstacle avoidance as well as positioning of the supporters (the players that do not handle the ball but try to reach a good position for support, pass interception, or defense).

4.2.2.1 Walking to a Position

There are two basic behaviors for walking to a position. First, *"go-to-point"* has the parameters *"x"* and *"y"* for the destination point, *"destination-angle"* for the orientation of the robot at the end, and *"max-speed"* for the maximum walk speed. As the rotation which is needed to reach the target angle is distributed over the whole distance to the target, it may happen that the robot walks backward.

Second, basic behavior *"go-to-point-and-avoid-obstacles"* uses the vision based obstacle model [29] to avoid obstacles on the way to the destination. Therefore, the robot has to walk forward to be able to detect the obstacles. The parameter *"avoidance-level"* defines how strict collisions shall be avoided. As it walks forward to its destination, it has no parameter for a target angle.

The option *"get-to-position-and-avoid-obstacles"* (cf. fig. 4.14a) combines these two basic behaviors. Far away from the destination, in state *"far-from-destination"*, *"go-to-point-and-avoid-obstacles"* is used. As this basic behavior has problems near the target and as a target angle has to be reached, in state *"near-destination"* *"go-to-point"* is used. The distance from which on no obstacles shall be avoided can be set with the parameter *"no-obstacle-avoidance-distance"*. At the destination in state *"at-destination"*, the robot

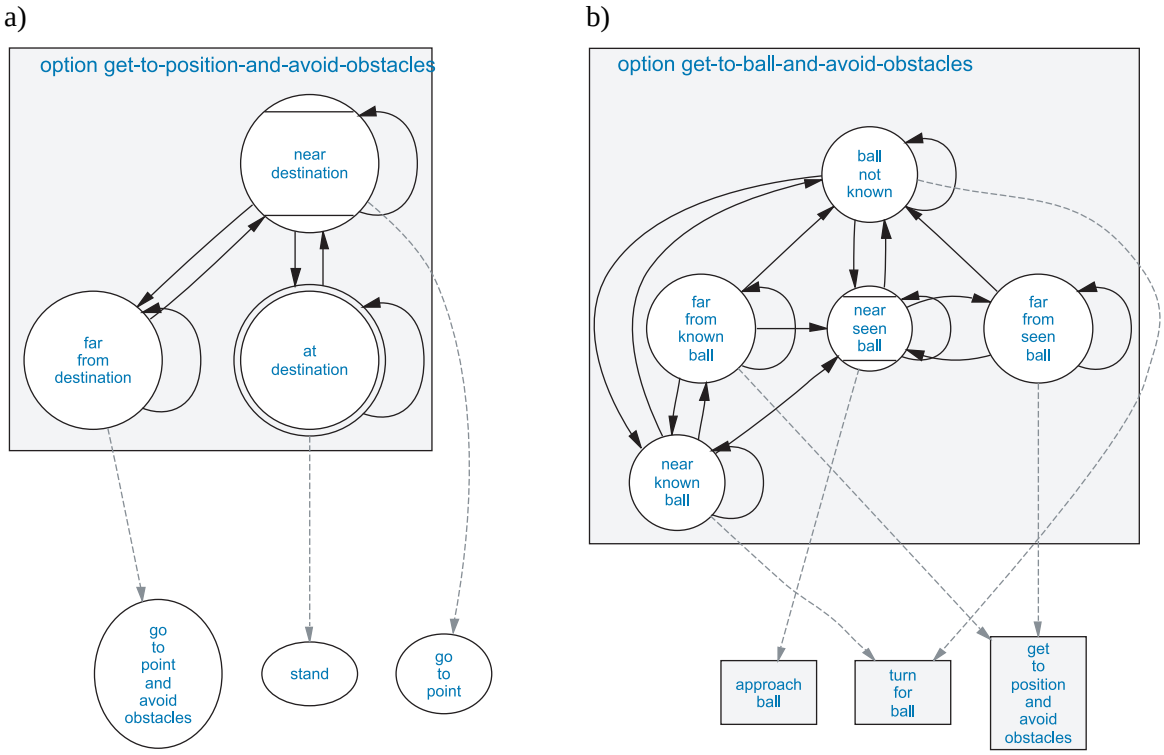


Figure 4.14: a) Option *"get-to-position-and-avoid-obstacles"* walks to a position avoiding obstacles on the way there. b) On top of that, *"get-to-ball-and-avoid-obstacles"* walks to the ball.

stops using the basic behavior *"stand"*.

4.2.2.2 Walking to a Far Away Ball

The ball handling behaviors do not perform any obstacle avoidance and are therefore only executed near the ball. For longer distances, option *"get-to-ball-and-avoid-obstacles"* (cf. fig. 4.14b) is used.

For the ball position, there is a distinction between "seen" and "known". A "seen" ball position is a position that was modeled from perceptions made by the own camera of the robot. A "known" ball position is derived from a ball that was either seen or, after a time of 5 seconds in that no ball was seen, from a ball position that was transmitted over the Wireless LAN by team mates (the "communicated" ball position). As the "seen" ball position is measured and modeled relative to the robot, it is independent from localization errors. Instead the "communicated" ball position contains both the localization errors of the sending and the receiving robot and is therefore much more imprecise. That's why the "known" ball position can only be used to walk approximately into the direction of the ball but not for exact positioning near the ball or even ball handling.

If the ball is seen and far away, in state *"far-from-seen-ball"* the option *"get-to-position-and-avoid-obstacles"* is executed at high speed. If the ball is not seen but known and far, the same option is used in *"far-from-known-ball"* at a medium speed. Near the seen ball, in state *"near-seen-ball"*, option *"approach-ball"* is chosen. If the ball is not seen but known in the near, option *"turn-for-ball"* searches the ball, as from the short distance the robot would see the ball if the communicated ball position was correct.

4.2.2.3 Positioning

The *GermanTeam* employed artificial potential fields for the positioning of the supporters on the field [41]. The basic behavior *"potential-field-support"* has the parameters *"x"* and *"y"* for the destination point as well as *"max-speed"* for the maximum speed. Inside, a potential field of superposed force fields with repelling forces from obstacles, the own penalty area, and the ball, tries to navigate the robot to the requested point without collision and without obstruction of the ball handling robot. At the same time the body of the robot is always oriented towards the ball.

Amongst others, the option *"position-supporter-near-ball"* (cf. fig. 4.15) makes use of that basic behavior. It tries to support a ball handling robot by staying near the ball to be available if the other robot loses the ball for some reason. Additionally, opponent robots are pushed away or obstructed in approaching the ball.

The parameters *"x"* specifies the desired relative x offset in field coordinates and *"y"* the distance in the y direction to the ball. The actual side (in y direction) is chosen in the initial state *"choose-side"*. If the ball is at the left border ($y > 80$ cm), the robot positions right to the ball, vice versa at the right border. In the center of the field the robot chooses the side on that it is already.

As there is very often a crowd of robots around the ball, especially in games against weaker teams, the ball is often not seen, leading to an imprecise ball model. Therefore, the supporting robots try to keep calm and move cautious in order to stay well localized. For that, in the states *"position-left-ball-seen"* and *"position-right-ball-seen"* the maximum speed of movement is set to 350 mm/s second minus 20 mm/s for every second that the ball was not seen. If the ball is not seen but known (see above in sect. 4.2.2.2), in the states *"position-left-ball-known"* and *"position-right-ball-known"*, the robots walk only half that fast as the communicated ball position is very erroneous near the ball. For the case that the ball is neither seen or known, there are two states for option *"turn-for-ball"* in order to continue on the previous side if the ball is found again.

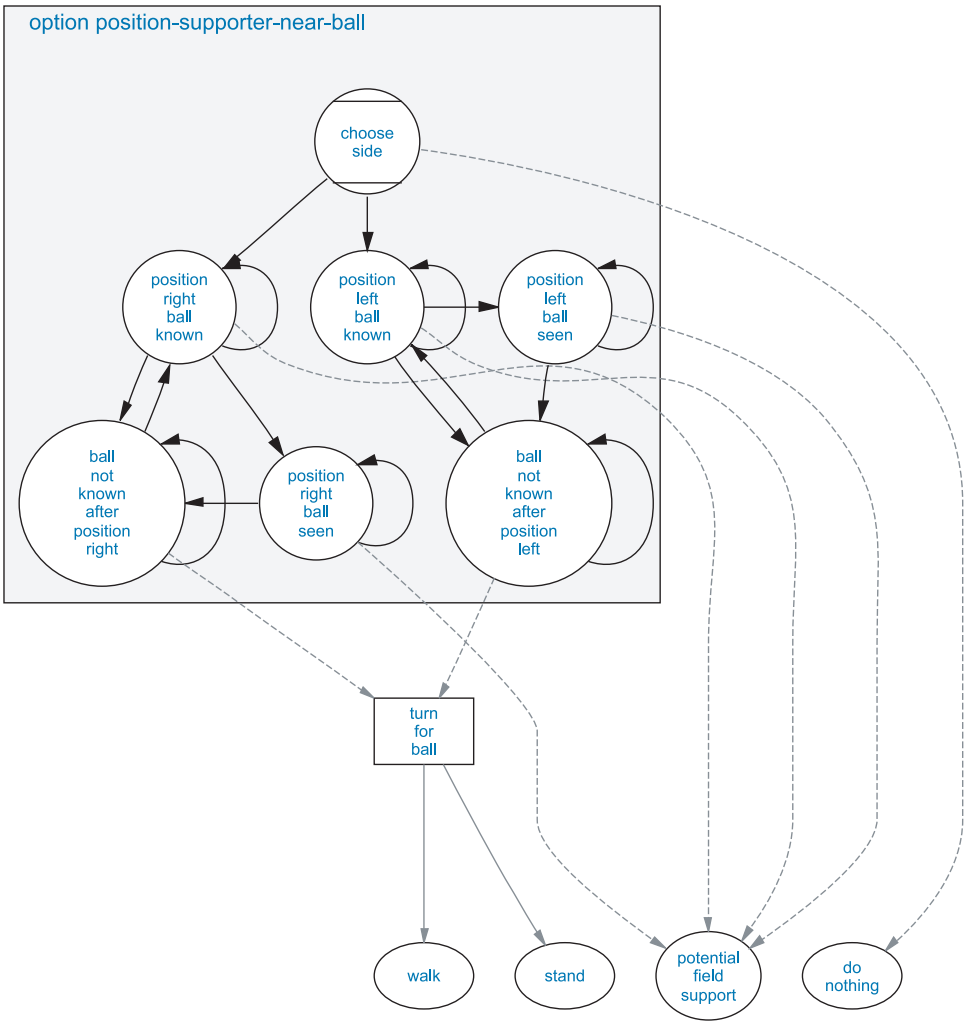


Figure 4.15: Option *"position-supporter-near-ball"* positions the robot near the ball. The speed is controlled depending on the reliability of the ball position.

4.2.3 Player Roles

The four robots on the field have different roles. The player with the number one is always the goalkeeper, the other three players change their roles dynamically. There is always only one robot at the same time that approaches the ball, the "striker". The "offensive supporter" positions in front of the ball or in the opponent half and the "defensive supporter" backs up from behind the ball and stays in the own half of the field.

4.2.3.1 Striker

The complete soccer playing behavior of the striker is implemented in option *"playing-striker"* (cf. fig. 4.16). In state *"get-to-ball"*, the option *"get-to-ball-and-avoid-obstacles"* (cf. sect. 4.2.2.2) is executed to

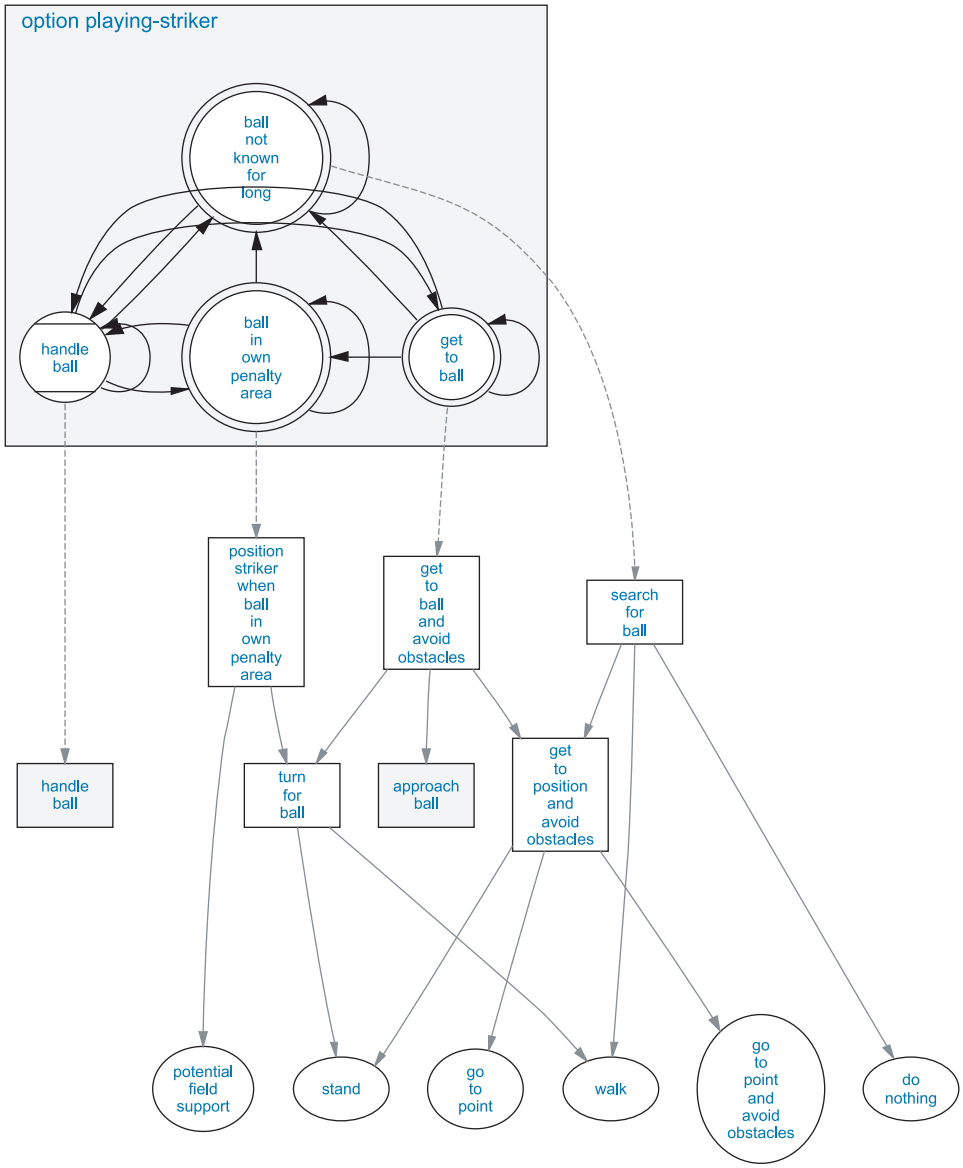


Figure 4.16: Option *"playing-striker"* implements a complete striker.

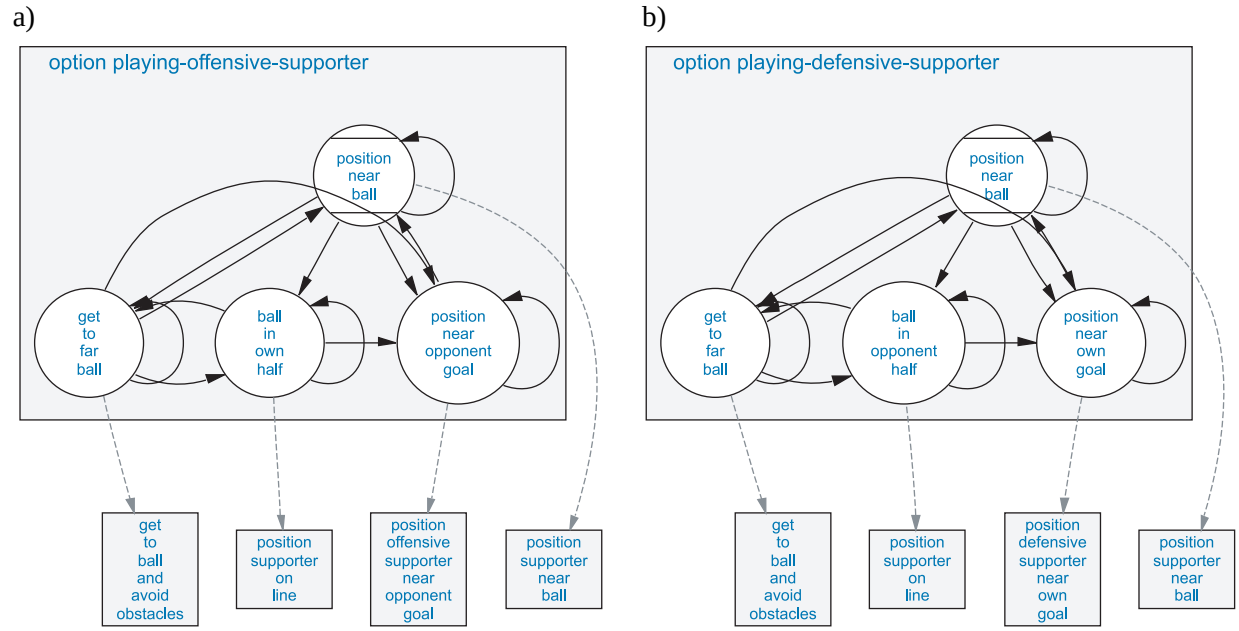


Figure 4.17: a) Option *"playing-offensive-supporter"* and b) *"playing-defensive-supporter"* decide where to position the robot.

approach the ball while avoiding obstacles on the way there. If the ball gets closer than 90 cm, in state *"handle-ball"* option *"handle-ball"* (cf. sect. 4.2.1.5) approaches and handles the ball without avoiding obstacles, as this would be disadvantageous. The back transition from *"handle-ball"* to *"get-to-ball"* is if the ball is farer away than 120 cm.

When the ball is inside the own penalty area (where the field players are not allowed to be in), in state *"ball-in-own-penalty-area"* the option *"position-striker-when-ball-is-inside-own-penalty-area"* positions the striker at the side of the penalty area, waiting for the goalie to clear the ball out of it.

Sometimes it happens that none of the four players of the own team is able to detect the ball for 12 seconds (for instance when two robots of the opponent team obstruct each other with the ball between them). Then, in state *"ball-not-known-for-long"* option *"search-for-ball"* walks along a fixed path between the left and right border of the field in order to redetect the ball. As also the supporters do that in other areas of the field, the whole field is covered and the ball is found again soon.

4.2.3.2 Supporters

The main tasks of the supporters is to position themselves well for pass interception, defense, and support of the striker. They cover the whole field in order to be able first at the ball if the ball is kicked out of a crowd.

At last, they try to stay away from the ball in order to not obstruct the striker.

The “offensive-supporter” stays most of the time in front of the ball and is implemented in option *“playing-offensive-supporter”* (cf. fig. 4.17a). If the ball is in the own half, in state *“ball-in-own-half”* the robot positions short behind the center line at the y position of the ball using option *position-supporter-on-line*, waiting for the pass. If the ball is inside the opponent half, in state *“position-supporter-near-ball”* the offensive supporter assists the striker by staying near the ball using option *“position-near-ball”* (cf. sect. 4.2.2.3). If the robot is still far away from the ball, in state *“get-to-far-ball”* the robot first walks there using option *“get-to-ball-and-avoid-obstacles”* (cf. sect. 4.2.2.2). If the striker plays the ball at the opponent border (which is detected through the position that the striker transmits over the WLAN), in state *“position-near-opponent-goal”* the supporter positions at the opposite corner of the penalty area using option *“position-offensive-supporter-near-opponent-goal”*, waiting for a pass or a failed kick of the striker.

Similar to that, the defensive supporter implemented in option *“playing-defensive-supporter”* (cf. fig. 4.17b) mostly stays behind the ball. When the ball is in the opponent half, in state *“ball-in-opponent-half”* positions in the middle of the own half at the y position of the ball using option *“position-supporter-on-line”*. If the ball is inside the own penalty area, the robot positions at the side of the penalty area opposite to the striker (state *“position-near-own-goal”* and option *“position-supporter-near-own-goal”*). Otherwise, it positions behind the striker to be there if the striker gets into difficulties.

Option *“playing-supporter-switch-roles”* selects between these two supporter options, depending on the role determined by the role negotiation process. It is executed from the initial state *“normal-playing”* of option *“playing-supporter”* (cf. fig 4.18) for the positioning of the supporters. In state *“ball-not-known-for-long”* option *“search-for-ball”* is executed if the ball is not known for more than 12 seconds. If a ball is going to roll fast closely along the robot towards the own goal, it is stopped in states *“block-left”* and *“block-right”* by jumping to the side. The actual analysis whether this could be successful is done in the ball locator module, storing the information in the ball model and providing it to the *XABSL* behaviors by the Boolean input symbols *“ball-rolls-by-left”* and *“ball-rolls-by-right”*.

In order to not loose the ball out of view when the striker kicks the ball somewhere, the striker notifies the other players on each kick through the Wireless LAN. Therefore, in all states of the ball handling options that prepare or perform a kick, the enumerated output symbol *“team-message”* is set to *“performing-a-kick”*. When the boolean input symbol *“another-teammate-is-performing-a-kick”* becomes true, in state *“intercept-before-kick”* the supporters stop positioning but look only at the ball using head control mode

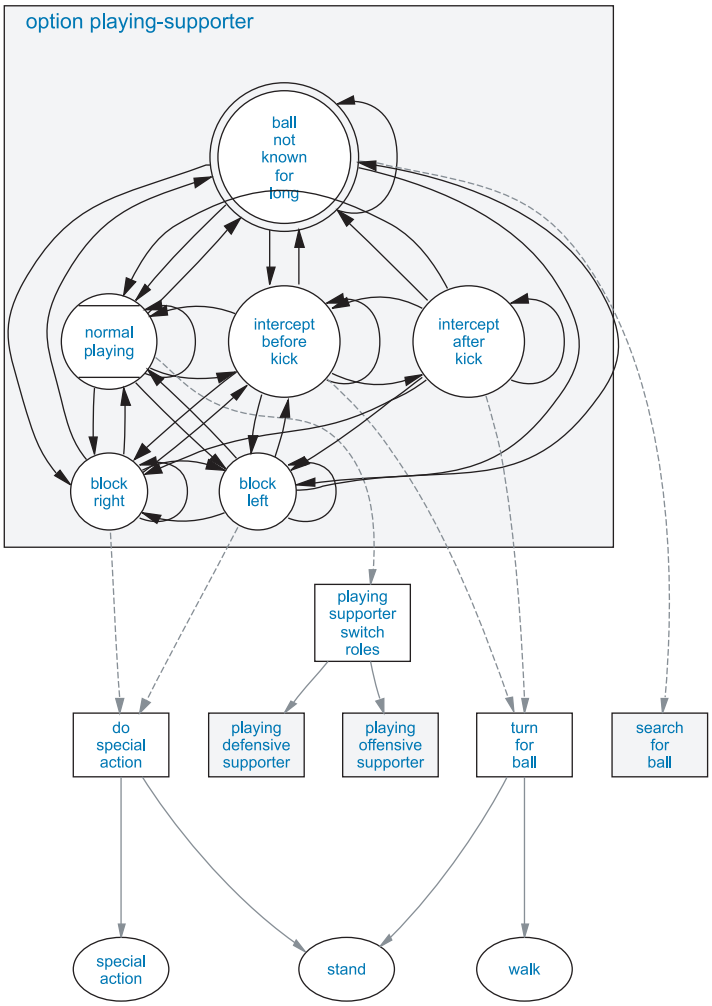


Figure 4.18: Option *"playing-supporter"* intercepts kicks from the own team and blocks kicks of the other team.

"search-for-ball" and turn themselves for the ball, using option *"turn-for-ball"* (cf. sect. 4.2.1.1). After the striker finished its kick, *"another-teammate-is-performing-a-kick"* is not true anymore. In state *"intercept-after-kick"*, the robot still turns for the ball until the ball does not roll anymore (low ball speed), the ball passed the robot forward (x position of ball greater than of the robot), the ball is not seen anymore, or after a timeout of 3 seconds.

4.2.3.3 Goalie

The *GermanTeam* had one of the best defenses in the RoboCup 2004 tournament, receiving only 8 goals compared to 65 goals scored by the team. This was achieved with a almost not moving goalie, standing at the right position for most of the time. As even small errors in the localization can make the robot believe that it

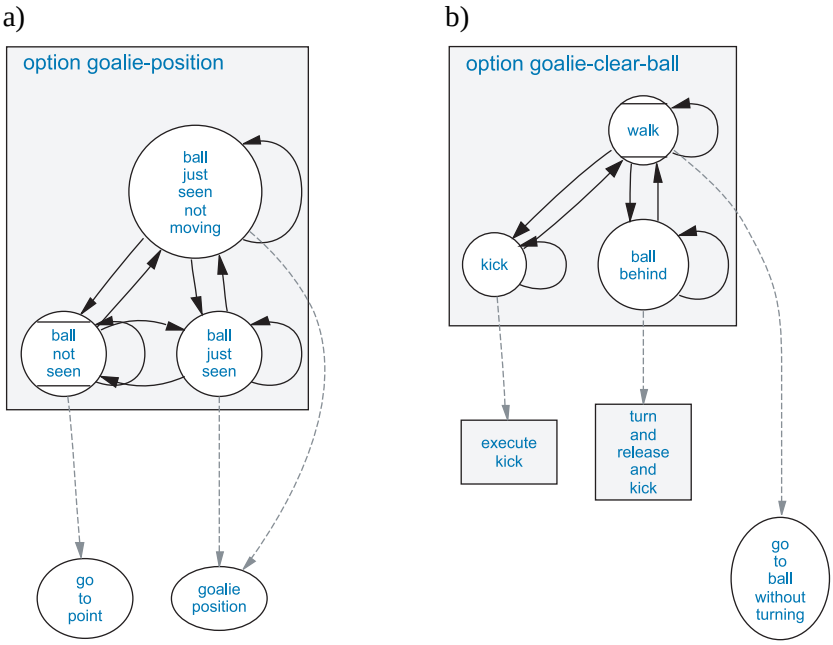


Figure 4.19: a) Option *"goalie-position"* positions the robot inside the goal. b) Option *"goalie-clear-ball"* tries to get the ball out of the penalty area.

is beside and not inside the own goal, the goalie behavior is much more dependent on good localization than the behaviors of the field players. The goalie behaviors have to support the localization with appropriate head movements and calm actions – it is very often a good strategy to let the goalie not move at all.

Option *"goalie-position"* (cf. fig 4.19a) makes use of the basic behavior *"goalie-position"*, which lets the robot position between the ball and the center of the own goal. To deal with errors in the localization, inside that basic behavior the robot's position is corrected using the odometry and the ball position – the robot uses the ball as a landmark and the odometry is trusted more than the position provided by the self localization. If the robot does not move (the basic behavior requests a *"stand"* motion), in the state *"ball-just-seen-not-moving"* the head control mode is set to *"search-for-ball"* (the head looks only at the ball), allowing for a better detection of fast balls. Otherwise, in state *"ball-just-seen"* the self localization is supported by setting the head control mode to *"search-auto"* (which scans also for landmarks). If the ball is not seen for 3.5 seconds, the robot walks to the center of the goal using basic behavior *"go-to-point"*.

Option *"goalie-clear-ball"* (cf. fig 4.19b) is responsible for the ball handling of the goalie. As there is mostly the striker and defensive supporter in the near, the task of the goalie is not to kick the ball very far (which requires strong and therefore dangerous kicks) but just to move it out of the penalty area. As there are often opponent robots that obstruct the goalie, no exact approaching of the ball is tried. Instead, in

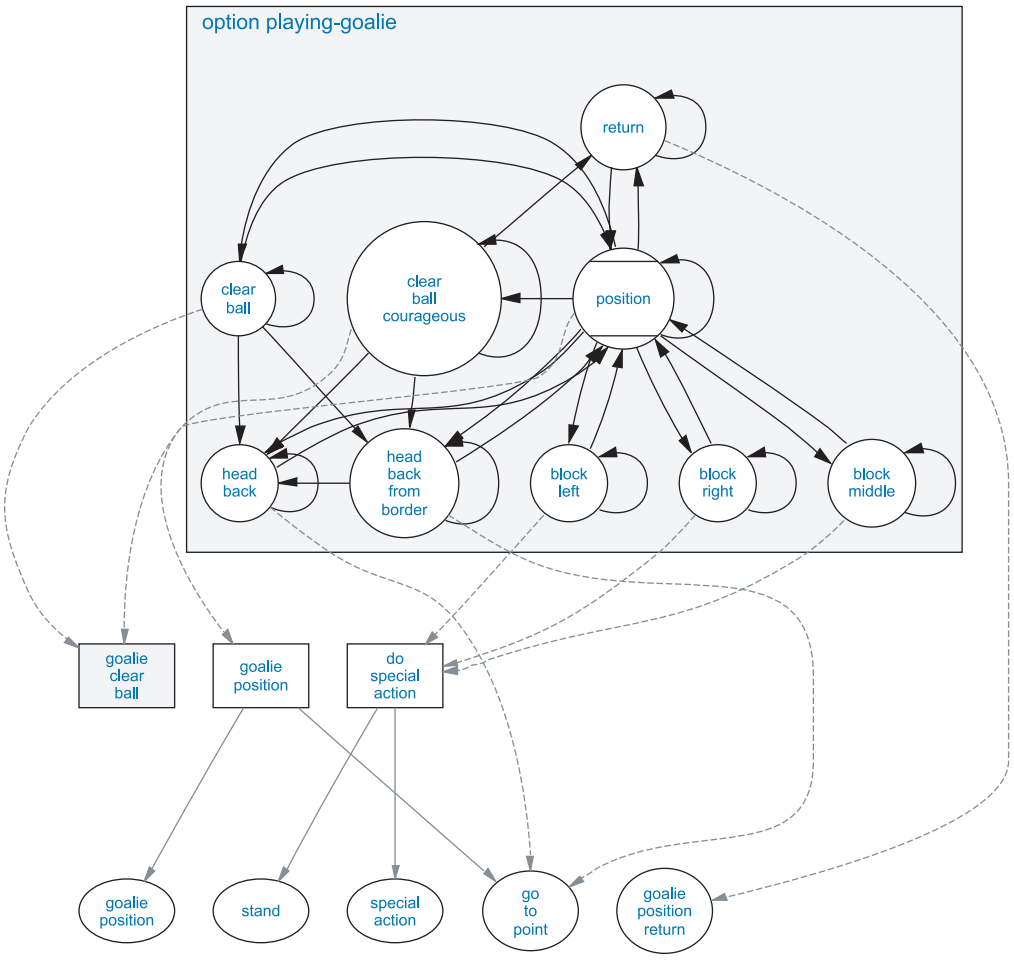


Figure 4.20: Option *"playing-goalie"* implements the goalkeeper behavior.

state *"walk"* the basic behavior *"go-to-ball-without-turning"* is used. Thereby the robot does not turn at all, which is faster than the normal *"go-to-ball"* basic behavior. If by chance the ball is in a good starting position for a kick (depending on a special kick selection table for the goalie, cf. sect. 4.2.1.4), in state *"kick"* a kick is performed. If it happens that the ball is behind the goalie (x position of the ball greater than the x position of the robot), in state *"ball-behind"* option *"turn-and-release-and-kick"* (cf. sect. 4.2.1.4) is used to clear the ball.

The complete goalkeeper behavior is implemented in option *"playing-goalie"* (cf. fig. 4.20). In the initial state *"position"*, option *"goalie-position"* is selected. If the robot is for some reason far out the own penalty area for some reason, it returns to it in state *"return"* using basic behavior *"goalie-posion-return"*, which is faster than *"goalie-position"*. Similar to the supporters (cf. sect. 4.2.3.2), the goalie blocks fast balls by

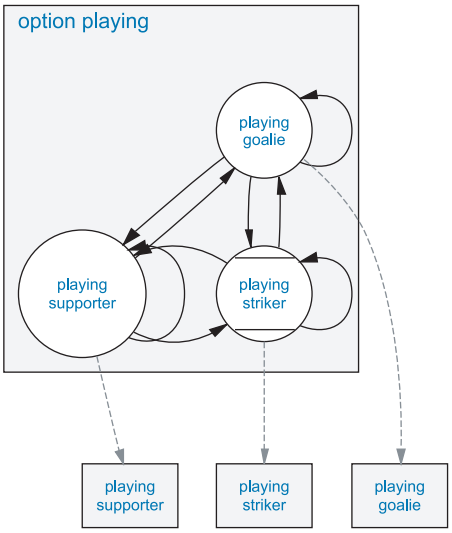


Figure 4.21: Option *"playing"* selects between the different roles.

jumping left, right, or ahead (states *"block-middle"*, *"block-left"*, and *"block-right"*).

Only when the ball is far inside the own penalty area (more than 20 cm over the line), in state *"clear-ball"* the option *"goalie-clear-ball"* is activated. There is already a transition back to state *"position"* when the ball is still inside the penalty area, 10 cm to the line. That's why it happens very often that the ball is far inside the penalty area and the goalie does not move, standing between the ball and the center of the goal. But this is a very good strategy, as opponent strikers only have a chance to get the ball across a well positioned goalie when the goalie makes an error and opens a gap. Additionally, it can provoke that opponent strikers are taken out due to the "goalie-pushing" rule or that continuous pushes from the strikers let the ball roll out of the penalty area by chance. Only when there are no obstacles (opponent players) in the near, in state *"clear-ball-courageous"* the goalie also clears a ball that is in the outer parts of the penalty area and it returns to *"position"* when the ball is 7 cm out of the penalty area.

If the goalie does not see a previously seen ball anymore for more than two seconds, it is very likely that the ball is at the side of the robot, where it can not be redetected by scanning around with the head. Therefore, in state *"head-back"* the robot walks backwards to the rear wall of the own goal for maximum four seconds, hoping to redetect a ball that is at the side of the robot. If the robot is at the field border besides the goal and does not see the ball anymore, in state *"head-back-from-border"* it first walks to the center of the goal line in order to not collide with one of the goal posts.

4.2.3.4 Dynamic Role Assignments

Option *"playing"* (cf. fig. 4.21) assigns the different roles to the four robots. As only a specially marked robot is allowed to be inside the own penalty area, player one is always the goalie. But the field players negotiate, which of them is the striker or a supporter. Therefore, all players transmit trough WLAN the time, how long they will approximately need to reach the ball. This time is computed such:

```
estimatedTimeToReachBall = distanceToBall / 0.2
  + 400.0 * fabs(angleBetweenBallAndOpponentGoal)
  + 2.0 * timeSinceBallWasSeenLast;
```

For every 10 cm to the ball it is assumed that the robot needs 500 ms to get there. The angle between the ball and the opponent goal is multiplied with 400 ms and added, preferring robots that are already behind the ball (no time is added) over robots that would have to grab the ball with the head or that would have to turn behind it (maximum 400 ms $\times \pi/2$ is added). In the last term, two seconds are added for every second that the ball was not seen, preferring robots that see the ball well.

For the role negotiations, the robot with the least estimated time to reach the ball is chosen to be the striker. To stabilize the decision, the player that is already the striker gets a time bonus of 500 ms. From the other robots, the robot with the higher x position (plus a bonus of 30 cm for the current offensive supporter) becomes the offensive supporter.

As an exception, if a supporter positions in front of the opponent goal (option *"position-offensive-supporter-near-opponent-goal"*, cf. sect. 4.2.3.2), it becomes immediately a striker if the ball is between the robot and the opponent goal.

If the WLAN does not work, a fallback with semi-fixed mappings from robot numbers to roles is applied: Player number two becomes striker if the ball is not far in the opponent half (x position of the ball less than 50 cm) and if the ball was seen in the last five seconds. Otherwise, it is a defensive supporter, staying in the own half. Players three and four become strikers when the ball is not far in the own half (x position of ball greater than -50 cm), otherwise they are offensive supporters. This can lead to situations (when the ball is in the center of the field) in that all three field players are strikers, which does not look very good.

The computed role is provided to the *XABSL* behaviors through the enumerated input symbol *"role"*. However, in option *"playing"* this role is not directly mapped onto the states for the different roles. For example, if the striker performs a kick or has the ball grabbed (the Boolean symbol *"ball.is-handled-at-*

the-moment" is true), option *"playing"* remains in state *"playing-striker"*. Additionally, if the supporters intercept a pass (option *"playing-supporter"* is not in one of its target states), there is also no transition to other states. This is helpful if a ball is kicked in the direction of a supporter. It becomes only a striker when the ball passed the robot or if the ball does not roll anymore, preventing the robot from running into the wrong direction and possibly pushing the ball back.

4.2.4 Game Control

The *GermanTeam* supports the RoboCup Game Manager to minimize human interaction during the games. This program is operated by a co-referee and sends via WLAN the state of the game (*initial*, *ready*, *set*, *playing*, *penalized*, or *finished*), the current score, the team color, and which team has kick-off to both teams. If the WLAN does not work for some reason, there is a sophisticated standardized interface to set these states manually through the buttons of the robot.

If all the game states would be implemented in one option, the number of transitions between states would be unmanageable high, as there are both transitions for messages from the game controller and button press events. Additionally, the *GermanTeam* added also transitions that are needed when the game controller is wrongly operated. That's why the implementation of the game control is distributed over three options: *"play-soccer"*, *"initial-ready-and-set"*, and *"initial-set-team-color"*.

The option *"play-soccer"* (cf. fig. 4.22) is the root option of the option graph. It has a state for the *"penalized"* game state where the robot does not move, a state for the *"finished"* game state where the option *"finished"* is executed (cf. sect. 4.2.5), and a state for the *"playing"* game state where the option *"playing"* (cf. sect. 4.2.3.4) is executed. All other game states are managed by the state *"initial-ready-and-set"*, executing the option with the same name. As the option *"initial-ready-and-set"* also executes a kick-off behavior when the *"playing"* message was received, *"play-soccer"* switches only from *"initial-ready-and-set"* to *"playing"* when *"initial-ready-and-set"* is in its target state, indicating that the kick-off behavior is finished.

The option *"initial-ready-and-set"* (cf. fig. 4.23) implements the game states *"initial"*, *"ready"*, and *"set"*, as well as the post-kick-off behavior. As the kick-off positions and the post-kick-off behaviors are different for own and opponent kick-off, there are always two states for all game states.

In the beginning, if there was a goal, in the state *"own-team-scored"* or *"opponent-team-scored"* the corresponding option performs a short happy or sad cheering move (cf. sect. 4.2.5). When these options

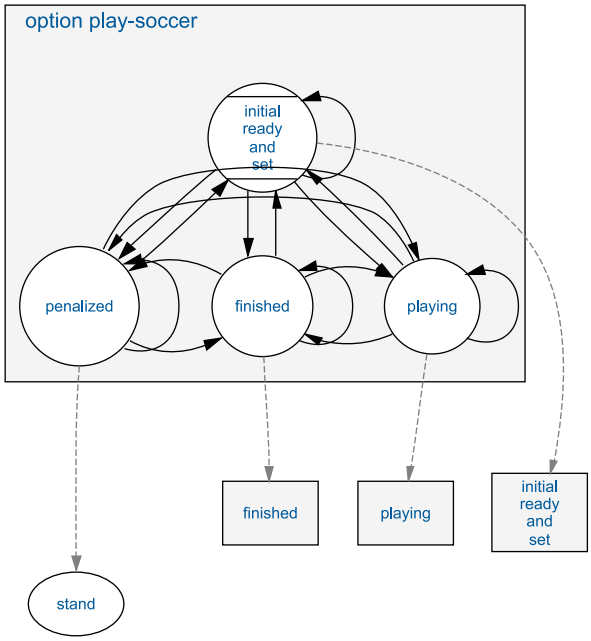


Figure 4.22: Option *"play-soccer"* is the root option of the option graph.

reach their target states, the option switches to the states for the *"ready"* game state.

In the *"ready"* states, the option *"go-to-kickoff-position"* lets the robots autonomously walk to their kickoff positions. These positions are read from input symbols to make them easy to configure. For own kick-off, one robot (robot four) is allowed to go to the center circle. If it gets close to that, in *"go-to-kickoff-position"* the state *position-exactly* becomes active, trying to position the robot very precisely and such that after the kick-off the robot can kick the ball straight ahead through the biggest gap between the opponents. Additionally, robot three positions at the center line close to the border. To avoid that opponent teams adapt to the *GermanTeam*'s kick-off strategies, there are different variants, which are selected randomly.

With the *"set"* message from the game manager or by touching the head button, the states for the *"set"* game state are reached. Before own kick-off, the option *"set-before-own-kickoff"* is executed. This option lets robot three, which positioned at the centerline, perform a different standing pose, allowing him a faster start after the kick-off.

After the *"playing"* message from the game controller or after a pressed head button, the states *"playing-after-own-kickoff"* and *"playing-after-opponent-kickoff"* become active, executing the corresponding options. In option *"playing-after-opponent-kickoff"* the target state is immediately reached for the goalie, robots three, and four. This lets the option *"initial-ready-and-set"* reach its target state *"playing"*, which again allows for a transition to *"playing"* in option *"play-soccer"*. But robot two keeps standing for

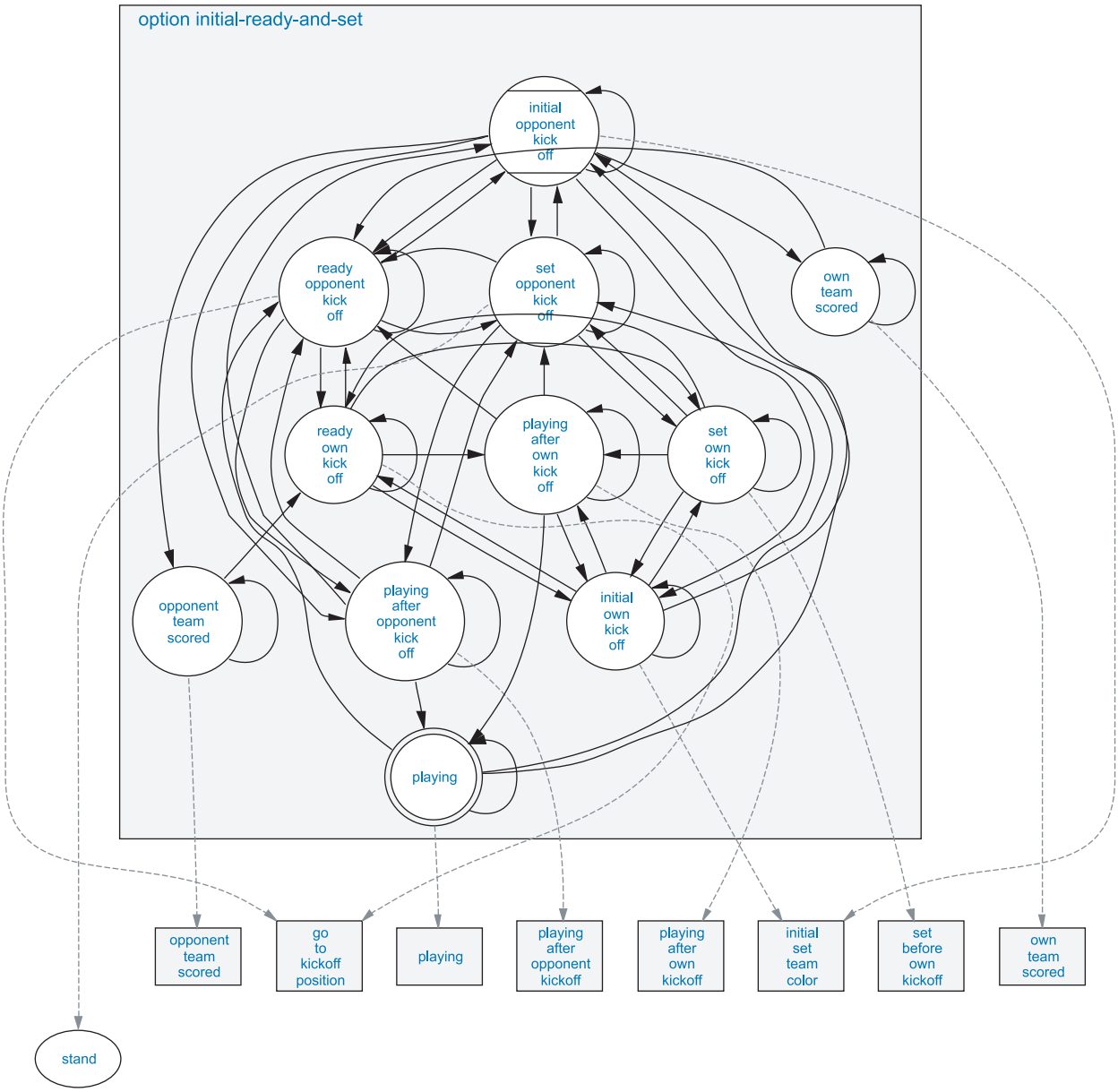


Figure 4.23: Option *"initial-ready-and-set"* is probably the most complicated looking one.

a) b)

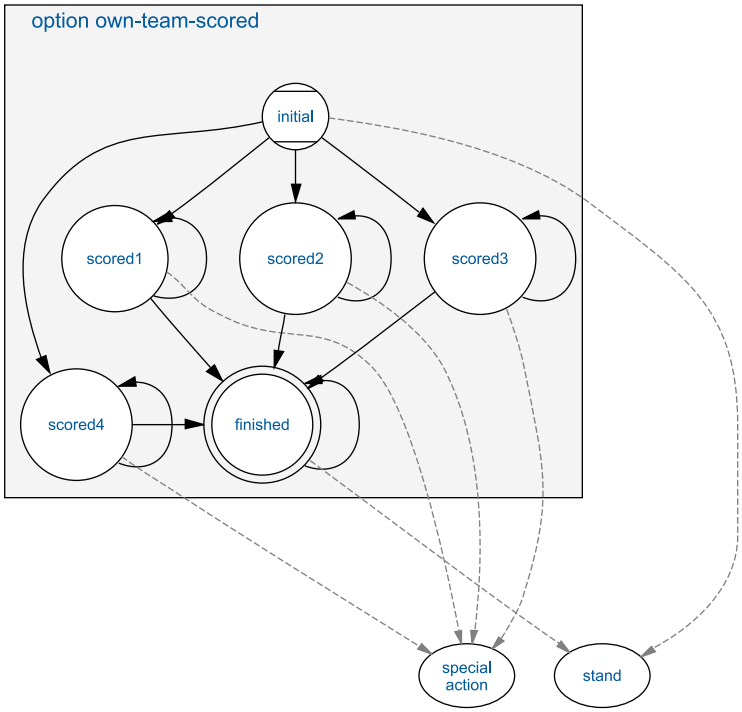


Figure 4.24: a) Option *"own-team-scored"* only chooses one out of four *"scored"* states, executes them for a few seconds, and then terminates. b) At the end of option *"finished"*, all robots walk to the center of the field for being also on the winner photo.

4.5 seconds to avoid that three field players run for the ball, possibly causing problems with the role negotiations. In the option *"playing-after-own-kickoff"*, the goalie and player two immediately start playing using the same target state mechanism. Player four performs a strong kick straight ahead. Player three runs blind with the "dash" walk type for 2 seconds along the border into the opponent half. If player four hits the gap between the opponent robots, player three can approach the ball at the opponent border before the opponent team does. However, this strategy worked well only against weaker teams.

If the robots are operated by hand, the option *"initial-ready-and-set"* is in the states for the *"initial"* game state at the beginning. Both execute the option *"initial-set-teamcolor"*, which allows for manual setting of team color through the back buttons.

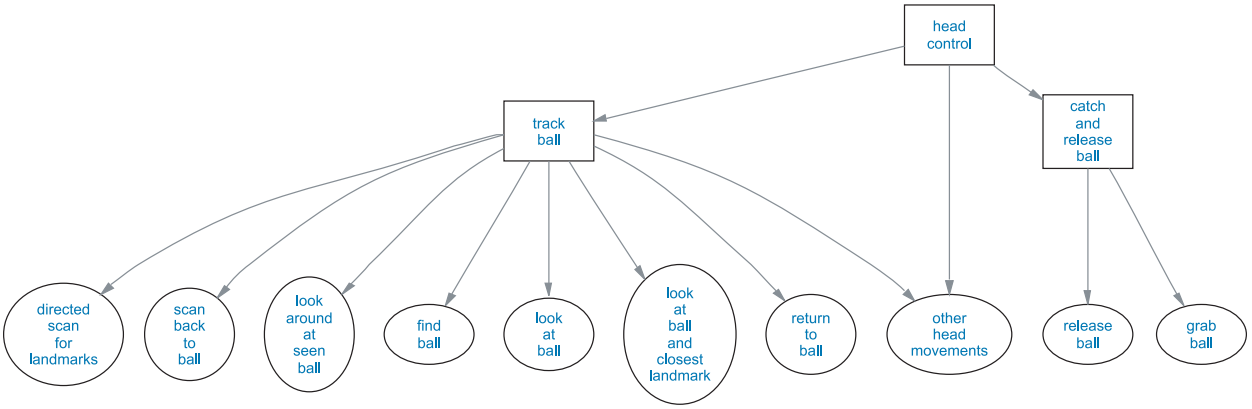


Figure 4.25: The option graph of the head control behaviors consists of only three options. The actual work is done in the basic behaviors.

4.2.5 Cheering and Artistry

The Sony Four Legged League is highly interesting to watch because the robots behave very life-like and the game is highly engaging. To make the games more enjoyable for the crowds, cheering (and crying) behaviors were implemented in addition to the wide range of kicks. After each goal, in option *"own-team-scored"* (cf. fig. 4.24a) one of four happy looking cheering motions is executed. After a few seconds, the option reaches its target state and the robots walk back to their kick-off positions. Accordingly, option *"opponent-team-scored"* selects between four annoyed and sad looking motions.

After the game, when the own team lost, in option *"finished"* the robots just let their heads hang down and behave sad. But when the own team won, the choreography is a bit more complex. All robots slowly walk to the center of the field. During this, every seven seconds, they stop walking and perform synchronously some cheering motions. After a while, all robots arrive in the center of the field and continuously perform headstands, which gives a good foreground for the winner photo (cf. fig. 4.24b).

Besides the cheering motions for the soccer games, many other demos and artistry choreographies were developed with *XABSL*.

4.3 Head Control with XABSL

Besides for behavior control, *XABSL* is also used by the *GermanTeam* for the control of the head movements in the *HeadControl* module. Although the head control behaviors are relatively simple, it turned out that the previous C++ implemented version of the state machine was very bad to handle. None of the developers really had an overview over the code and the whole thing was very difficult to debug. So the use of *XABSL*

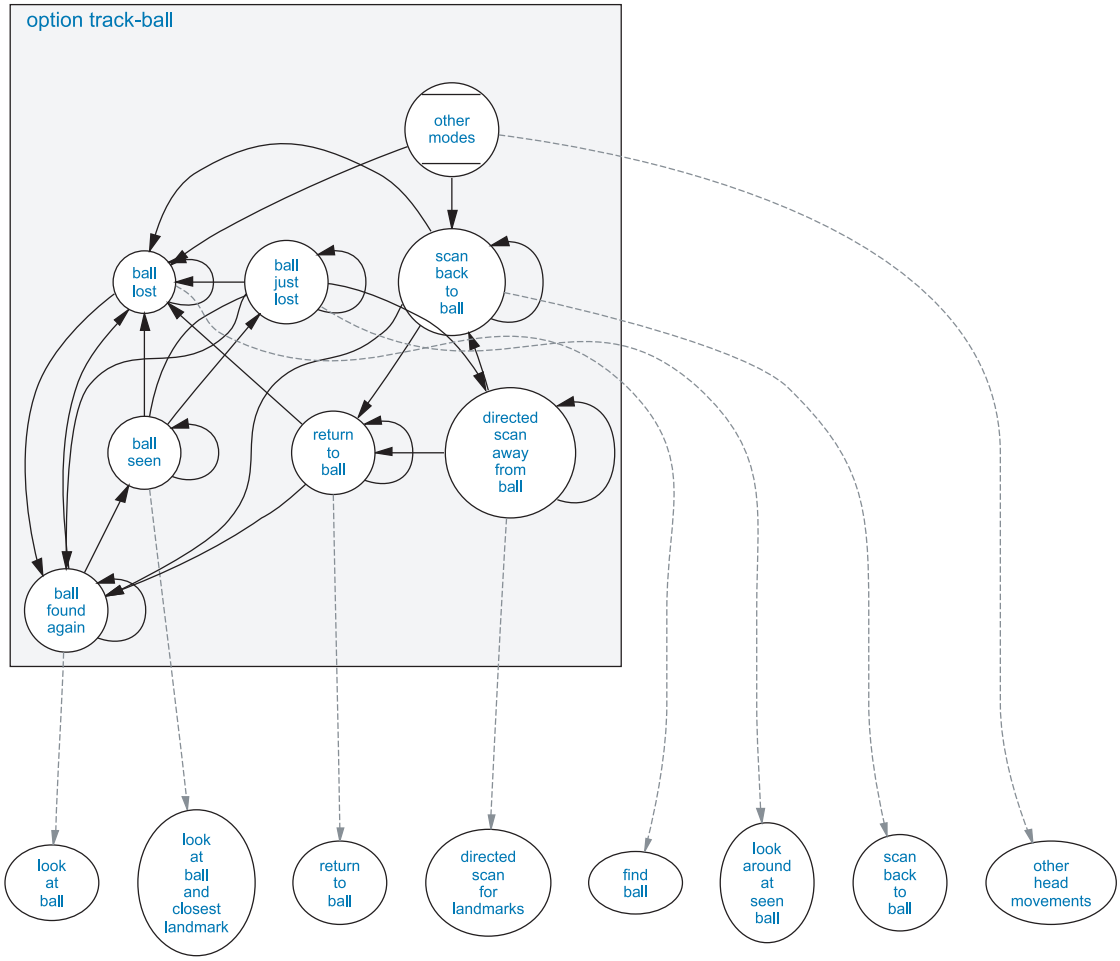


Figure 4.26: Option *"track-ball"* is responsible for the ball tracking head control modes *"search-for-ball"* and *"search-auto"*.

in that module was mainly motivated from software engineering aspects.

Figure 4.25 shows the admittedly small option graph. Most of the actual work is done in the basic behaviors and the *XABSL* options are only responsible to activate them in the right time. The main input for the decisions are the requested head control mode, the ball model, states of the walking engine, and internal states of the head path planning component.

The root option *"head-control"* only maps the head control mode to the subsequent options. In option *"catch-and-release-ball"* it is made sure that a caught ball is released at the right position in walk cycle, which is needed in the behavior control option *"turn-and-release"* (cf. sect. 4.2.1.3).

The only complex option in the head control is *"track-ball"* (cf. fig. 4.26). It implements the two ball tracking head control modes *"search-for-ball"* (the robot only looks at the ball) and *"search-auto"* (the

robot looks at the ball and regularly scans around for landmarks to improve the obstacle model and self localization). If previously another head control mode than these two was requested, there is a transition from the initial state *"other-modes"* to *"scan-back-to-ball"* if the ball was seen in the last second or to *"ball-lost"* otherwise. In *"ball-lost"*, the basic behavior *"find-ball"* scans around along a fix path to redetect the ball. If the ball is seen again, in state *"found-ball-again"* the head is strictly centered to the ball for 500 ms in order to stabilize the ball tracking. After that, in state *"ball-seen"* basic behavior *"look-at-ball-and-closest-landmark"* looks at the ball and, if possible by keeping the ball in sight, the next landmark. If during this the ball is not seen anymore for more than 500 ms and the ball was close before, in state *"ball-just-lost"* the basic behavior *"look-around-at-seen-ball"* scans in the near of the previously seen ball. If that fails, the whole area of view is again scanned in *"ball-lost"*. If in *"ball-seen"* the ball gets lost and was seen far away before, there is a direct transition to *"ball-lost"*.

For the *"search-auto"* mode, there is a transition from *"ball-seen"* to *"directed-scan-away-from-ball"* if the ball was just seen consecutively for more than 1 second or if the ball rolls away from the robot. The robot then in basic behavior *"directed-scan-for-landmarks"* scans for the next landmarks for maximum 800 ms. After that, it scans back to the ball in state *"scan-back-to-ball"* using basic behavior *"scan-back-to-ball"*. If the head arrived at angles where it should see the ball again, the ball is assumed to be lost and searched again in *"ball-lost"*. If during the whole scanning procedure the head control mode changes back from *"search-auto"* to *"search-for-ball"*, the head immediately returns to the ball in state *"return-to-ball"*.

To conclude, it would not have been necessary to employ *XABSL* for head control. Nevertheless, it made the development process more straightforward and manageable.

4.4 XABSL in the ASCII Soccer Simulator

The ASCII Soccer environment was developed by Tucker Balch around 1995 [10]. In this very simple soccer simulation the field is displayed on a text terminal (cf. fig. 4.27. It is 78 characters long and 21 lines wide. Two teams of four players each are displayed with "$>$" and "$<$" characters. They try to get the ball ("o") into the opponent goal. The complete right hand and left hand sides of the field are the goals. A game lasts until one team has scored 7 goals.

The players are able to directly sense their position on the field, the rough direction of the ball (N, NE, E, SE, S, SW, W, or NW), and the objects (ball, players, wall) which are in the direct neighborhood (on one of the 8 neighbor places) of the player. The action set of the agents is very limited: They can either move

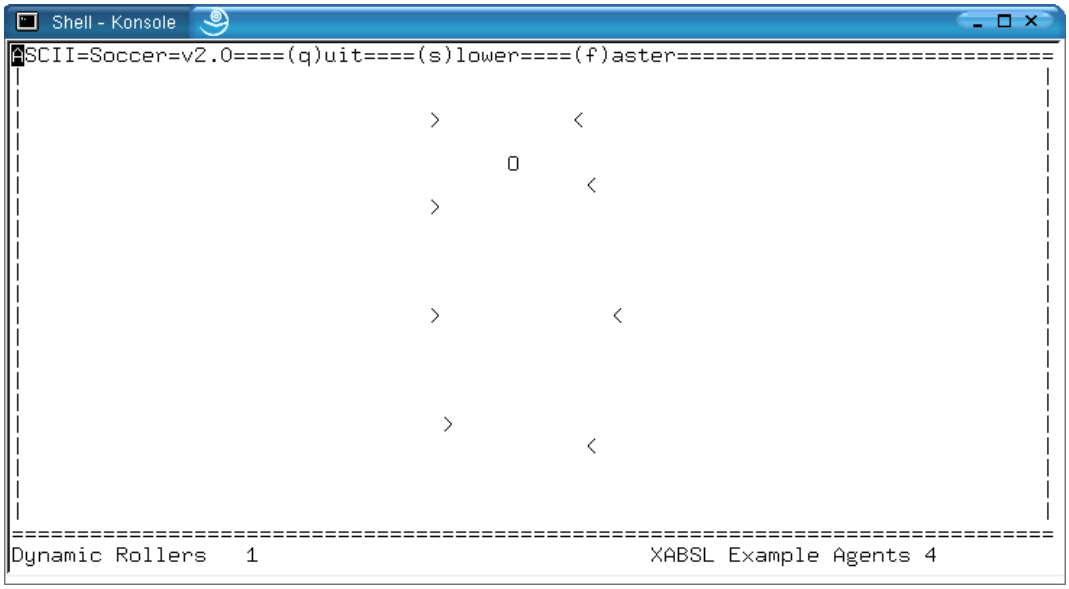


Figure 4.27: A scene from an ASCII Soccer game shortly after a kick off. The team “Dynamic Rollers” (“>”) plays from left to right and the *XABSL* example agent team (“<”) from right to left.

to one of the eight neighboring places or kick. The simulation is not deterministic. From crowds kicking the ball at the same time, it is not predictable where the ball goes to.

Due to the accessibility to a nearly complete world model and the limited action set it was possible to develop a simple and easy to understand *XABSL* agent team in short time. The root option switches between a defender behavior which positions behind the ball, a striker behavior that positions in front of the ball, and a midfielder behavior that is responsible for ball handling. These roles are assigned dynamically. The two players which are closest to the ball are the midfielders, the other two a defender or a striker depending on which of them is closer to the own goal. The ball handling selects between dribbling (when no team mate is in front of the player) and passing. A complete documentation of the behaviors can be found at the *XABSL* web site [45].

This simple implementation was able to win against all teams except one that were available at the ascii-soccer home page [10].

The ASCII Soccer *XABSL* example implementation can be downloaded together with the language definitions, the tools, and the *Xabsl2Engine* for free from the *XABSL* web site [45].

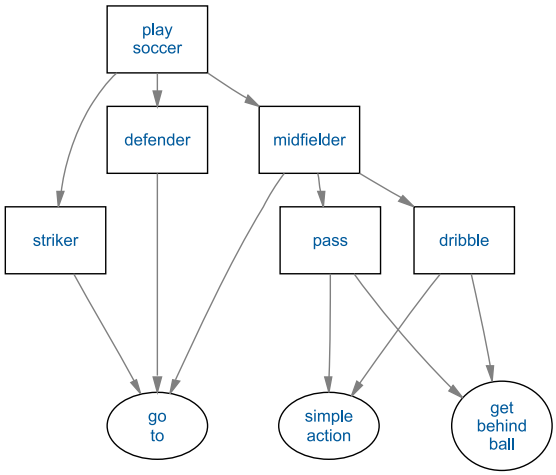


Figure 4.28: The option graph for the ascii-soccer *XABSL* example agent team.

5 Conclusion and Future Work

This thesis introduced the *Extensible Agent Behavior Specification Language* as an XML dialect that allows to conveniently develop the behavior of autonomous agents. The advantage of the underlying layered state machine architecture was shown. Together with all its supporting components, the *XABSL* system proved to be a powerful tool for behavior engineering.

It was shown how the *GermanTeam* successfully employed the system to develop complex team behaviors for RoboCup competitions in the Sony Four Legged League resulting in top positions at the RoboCup German Open in the last four years and the win of the RoboCup world championship 2004.

The language and the code library *XabslEngine* are independent from the software platform that the *GermanTeam* uses. It is relatively easy to employ *XABSL* on other robotic platforms. The *XABSL* schema files, the *XabslEngine* library, and the tools are open source and publicly available at the *XABSL* web site [45].

5.1 Results at RoboCup Competitions

The *Aibo Team Humboldt* won the German Open 2001 with a flat state machine approach (cf. fig. 5.1a). Afterwards a layered state machine approach was developed for the participation of the *GermanTeam* at the RoboCup 2001 competitions in Seattle. The team did not reach the quarter finals there (cf. fig 5.1b).

The first version of *XABSL* was used by the *Aibo Team Humboldt* at the German Open 2002 in Paderborn. Although the team scored more than one third of all goals scored in the tournament, it unfortunately was beaten by the *Darmstadt Dribbling Dackels* 0:1 in the final (cf. fig 5.1c). At the RoboCup 2002 competitions in Fukuoka, the *GermanTeam* finished the round robin being second in the group (cf. fig 5.1d). In the quarter final, the team had to compete against previous year's world champion UNSW and lost. But there were only two other teams that scored a goal against CMU, the 2002 year's world champion, namely Tokyo in the round robin and UNSW in the final. Only one other team, could score against UNSW, this was CMU in the final.

a)

Round Robin	
ATH – Darmstadt Dribbling Dackels	1 : 0
ATH – S.P.Q.R. Legged	4 : 0
ATH – Les 3 Mousquetaires	2 : 1
Final	
ATH – Darmstadt Dribbling Dackels	5 : 1

b)

Round Robin	
GermanTeam – Baby Tigers	1 : 4
GermanTeam – UNSW	0 : 11
GermanTeam – UW Huskies	3 : 0

c)

Round Robin	
ATH – Les 3 Mousquetaires	3 : 0
ATH – Bremen Byters	4 : 0
ATH – Darmstadt Dribbling Dackels	1 : 0
ATH – Microsoft Hellhounds	3 : 0
Final	
ATH – Darmstadt Dribbling Dackels	0 : 1

d)

Round Robin	
GermanTeam – S.P.Q.R. Legged	5 : 0
GermanTeam – Araibo	4 : 0
GermanTeam – CMU	1 : 3
GermanTeam – Georgia Tech	4 : 1
Quarter Final	
GermanTeam – UNSW	1 : 6

e)

Round Robin	
ATH – Dynamo-Pavlov Uppsala	6 : 0
ATH – Les 3 Mousquetaires	4 : 0
ATH – SPQR-Legged	3 : 1
Quarter Final / Semi Final / Final	
ATH – Bremen Byters	3 : 0
ATH – Microsoft Hellhounds	3 : 0
ATH – Darmstadt Dribbling Dackels	1 : 2

f)

Round Robin	
GermanTeam – Austin Villa	9 : 0
GermanTeam – UTS Unleashed!	2 : 2
GermanTeam – UPennalizers	3 : 1
GermanTeam – Asura	5 : 0
Quarter Final	
GermanTeam – CMU (x:x+1)	2 : 2

g)

Round Robin	
ATH – Hamburg Dog Bots	4 : 2
ATH – Les 3 Mousquetaires	12 : 0
ATH – Microsoft Hellhounds	2 : 1
Quarter Final / Semi Final / Final	
ATH – S.P.Q.R. Legged	15 : 0
ATH – Hamburg Dog Bots	2 : 1
ATH – Darmstadt Dribbling Dackels	2 : 1

h)

Round Robin	
GermanTeam – UNSW	4 : 2
GermanTeam – Team Chaos	13 : 0
GermanTeam – ASURA	6 : 1
GermanTeam – Georgia Tech	12 : 0
GermanTeam – Baby Tigers	7 : 0
Quarter Final / Semi Final / Final	
GermanTeam – CMU	9 : 0
GermanTeam – NUBOTS	9 : 2
GermanTeam – UTS-Unleashed	5 : 3

Figure 5.1: Competition results. a) *Aibo Team Humboldt* at the German Open 2001 in Paderborn. b) *GermanTeam* at RoboCup 2001 in Seattle. c) Aibo Team Humboldt at the German Open 2002 in Paderborn. d) *GermanTeam* at RoboCup 2002 in Fukuoka. e) Aibo Team Humboldt at the German Open 2003 in Paderborn. f) *GermanTeam* at RoboCup 2003 in Padova. g) Aibo Team Humboldt at the German Open 2004 in Paderborn. h) *GermanTeam* at RoboCup 2004 in Lisbon.

At the German Open 2003, all four members of the *GermanTeam* used *XABSL* for behavior description. Three of them were placed first, second, and third. Again the *Aibo Team Humboldt* scored more than one third of all goals scored in the tournament and again it unfortunately lost the final in the penalty shoot out against the *Darmstadt Dribbling Dackels* (cf. fig 5.1e). At the RoboCup competitions 2003 in Padova, the *GermanTeam* finished the round robin as winner of its group, even beating the later runner-up, the UPennalizers. In the quarter final, the *GermanTeam* lost in a 29 minutes penalty shootout against CMPack'03. However, the *GermanTeam* won the RoboCup Challenge with 70 out of 72 possible points, the behaviors for the challenge being also written in *XABSL*. The results of the games are shown in fig. 5.1f.

From 2003 to 2004, the behaviors of the *Aibo Team Humboldt* and the *GermanTeam* improved most compared to previous years. As the team could rely on a fully developed software architecture and perception and motor control capabilities, it had much time to enhance and fine-tune the behaviors of the robots. This lead to a win of the *Aibo Team Humboldt* at the German Open 2004 (cf. fig. 5.1g). Again, all members of the *German Team* used *XABSL* for behavior engineering. Interestingly, the teams were the better, the less other methods for decision making besides *XABSL* they implemented. The newcomer team *Hamburg Dog Bots*, who used the 2003 code release of the *GermanTeam*, almost only improved the *XABSL* high level behaviors and became third. Then, in the RoboCup world championships 2004 in Lisbon, the *GermanTeam* finally became world champion, winning all its games clearly and having the second highest goal difference after the runner up *UTS*.

The performance of the *Aibo Team Humboldt* and the *GermanTeam* improved from year to year. On the one hand, this is due to the enhanced perception, modelling, and motor control capabilities but, on the other hand, significantly due to the advances made in the behavior control architecture and the implemented behaviors itself.

Looking at the international RoboCup competition results of the *GermanTeam*, one can see a significant increase in the accumulated goal difference. From 4:15 in 2001, it went up to 15:10 in 2002, 21:6 in 2003, and 65:8 in 2004.

5.2 Future Work

XABSL proved to be a powerful tool for fast and efficient behavior engineering of autonomous agents. It helped the *GermanTeam* to become the 2004 RoboCup world champion and there are bright prospects for *XABSL* to be the basis for further behavior control developments of the team as well as for other teams who

will probably use the German Team's 2004 code release for further developments.

The *XABSL* architecture, language, and tools are well documented and there is a good chance that other members of the *Aibo Team Humboldt* or members of the *GermanTeam* will continue the work. From the author's point of view, there are potentials for improving the performance at RoboCup competitions in near all fields of information processing: perception, world modelling, motor control, behavior implementations, and the behavior architecture.

The less uncertainty there is in the data achieved from perception and world modelling processes, the more advanced and complex behaviors can be developed. For instance, intercept behaviors can be developed only if there is an accurate and reliable model of the ball speed. More deliberative passing and positioning behaviors require a stable model for the positions of the other players.

Faster and more accurate motor control programs decrease the uncertainty in the performed actions and are a general advantage in RoboCup competitions. The best team strategies are useless if robots of the opponent team are able to walk double speed.

5.2.1 Possible Extensions to XABSL

People who start working with *XABSL* are most bothered about the rigid limitations in the allowed data types for symbols, functions, and parameters. For example, the type of option and basic behavior parameters has always to be "decimal", forcing developers to use decimal `0` for `false` and `1` for `true` if they want to pass Boolean parameters to options or basic behaviors. This restrictions are due to the decision not to write an own *XABSL* compiler but to use XML Schema for validation. But it would be possible to allow more data types in several contexts.

It would be much work, as the Schema definitions with the ID constraints, the XSLT style sheets for code and documentation generation, the *XabslEngine*, and debug tools would have to be extended.

The generation of the HTML documentation containing SVG graphs for state machines, option graphs, and decision trees helps the developers very much to intuitively see what their behaviors do. But the work with *XABSL* could be much more convenient if there would be a graphical editor. The main problem in writing an editor from scratch or in adapting an existing state chart editor is the complexity of the transitions between states. For example, the graph in figure 3.2 is a strong simplification of the documented option. There is shown maximum one edge from one state to another, although there can be many reasons for a transition between two states, resulting from complex hierarchical decision trees. Standard state chart editors

do not have this problems as there are only simple Boolean predicates allowed for transitions between states.

Before writing an *XABSL* editor, one needs a good idea how to cope with the hierarchical decision trees and multiple transitions between states. Another possibility would be the remove the concept of the decision trees from *XABSL*, defining only simple Boolean predicates for transitions between states. But this would restrict the expressiveness of the language very much.

5.2.2 The Double Pass Architecture

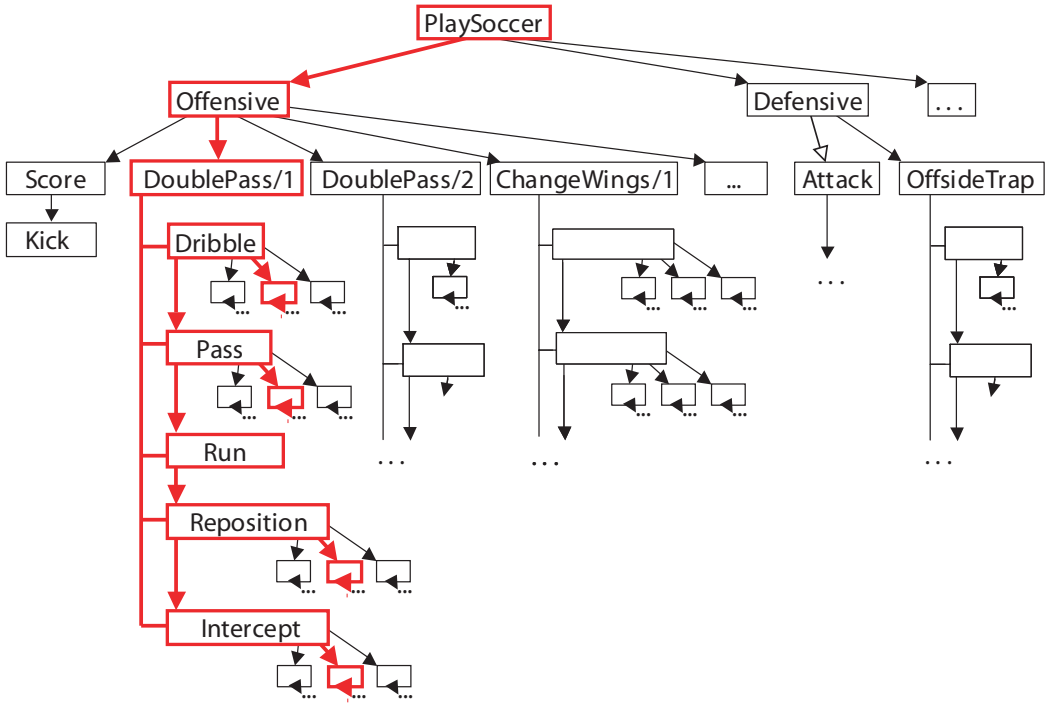
The camera of the Aibo ERS7 provides images every 33 ms (30 Hz). At the moment, the perception routines, the world modelling, and the behavior programs in the software architecture of the *GermanTeam* are executed in the same thread (system process *Cognition*). As the joints of the robot have to be controlled every 8 ms (125 Hz), the concurrent system process *Motion* executes the motor control programs. Almost all of the processing time of the *Cognition* process is used by the image processing and self localization programs. Up to now, the behavior control programs are executed for less than 1 ms. (The execution time is 0, but the time can be measured with a resolution of only 1 ms.) All high-level planning algorithms, the negotiations, and strategy decisions are executed together with the low-level behaviors within one execution of the option graph.

In future behavior control developments, it might happen that some planning algorithms are much more time consuming than the algorithms that are used at the moment. As soon as these algorithms are so time consuming that the whole image processing frame rate cannot be guaranteed, it might be useful to apply the *double pass architecture* [16].

In this architecture *options* similar to the ones in *XABSL* are ordered hierarchically in a rooted acyclic graph, the option tree (cf. fig 5.2). Different from *XABSL* options where arbitrary transitions between states are allowed, options in the double pass architecture are either *choice options* that choose between a set of subordinated options based on a utility or *sequence options* that perform subsequent behaviors in a fixed sequence (script).

The planning and decision making is distributed over two passes: The *deliberator* pass (cf. fig 5.2a) is responsible for the choice of intentions and for long-term planning. It sets up hierarchical plans and marks parts of the option tree as “intended”. The deliberator plans far enough into the future so that no time problems arise when it is not executed often enough. The *executor* pass (cf. fig 5.2b) is responsible for time critical decisions. It focuses only on the options that were marked as “intended” by the deliberator, what guaranties that only a few and little time consuming decisions are made in order to satisfy real-time

a)



b)

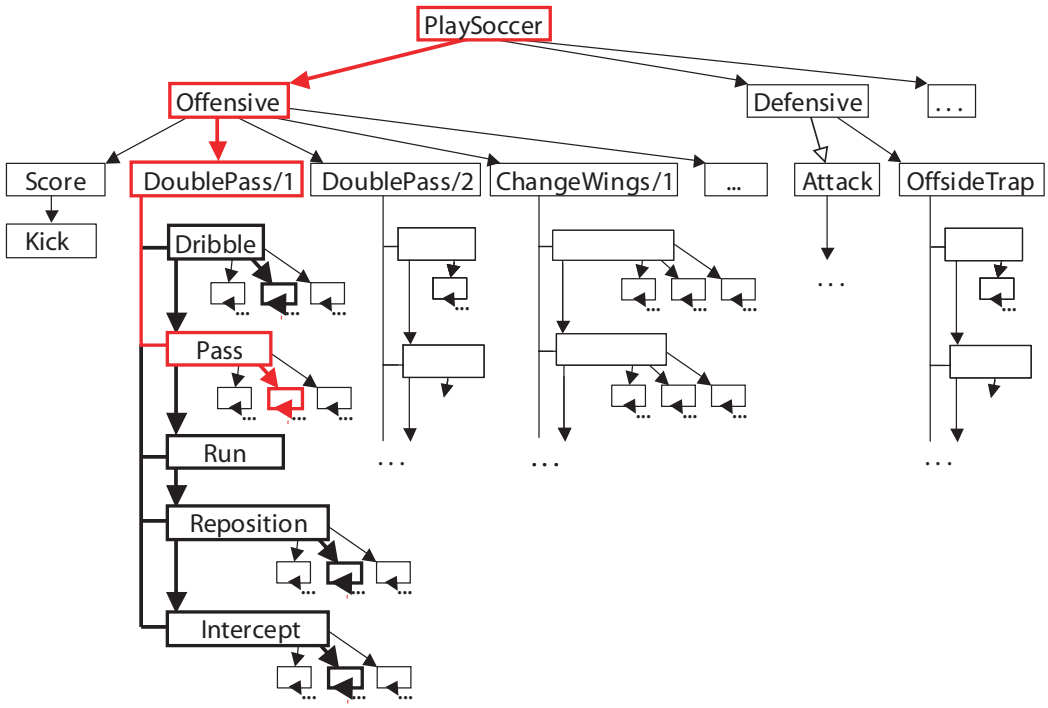


Figure 5.2: An option tree in the double pass architecture (from [16], example from the simulation league team *AT-Humboldt*). a) The intention subtree marked by the deliberator pass. b) The activity path of the executor.

conditions.

Note that there is no separation of the option tree into “deliberative” behaviors near the root of the tree and “reactive” behaviors near the leaves. Both the deliberator and the executor work on the same options.

There are two ways to synchronize this two passes: they can work in parallel or in sequence. Parallel work means that the deliberator and the executor run in two concurrent system processes. But on the Aibo system platform *OpenR* it turned out that additional processes and the resulting inter-process communication are very expensive. Therefore, it might be better to execute the two passes in the same system process in sequential order. In this case, the deliberator pass would need to be interrupted when real-time demands require the executor to be executed. After that the deliberator would continue its execution where had been interrupted before.

If the double pass architecture is applied to the Sony Four Legged League, the *XABSL* language and the *XabslEngine* would have to be adapted to it. For this purpose, it is possible to rely on the work of the ATH simulation league team at the artificial intelligence lab [11], because they also developed an XML based behavior description language following the double pass architecture.

A future goal might also be to unify *XABSL* and the work that was done in the simulation league team of the lab.

5.2.3 Learning Basic Capabilities

In the past, many ball handling behaviors were fine-tuned manually. Although the *XABSL* system provides a short change-compile-test cycle, this is a time consuming task. Besides, it is difficult to tune manually different parameters that are not independent from each other.

Many low-level behaviors such as ball approaching, ball grabbing, or turning around the ball could be improved with *machine learning* methods (cf. [52]). But a problem could be to find good performance measures or reward functions for the tasks. For instance when trying to grab a ball, it is difficult to measure whether the action was successful. Moreover, the number of training iterations is very limited. As there is no simulator for the Aibo robot that could satisfyingly simulate such low-level behaviors, the experiments have to be done with real robots, which is very time-consuming.

For some learning tasks *reinforcement learning* (cf. [68]) could be used. Stone [67] has shown how to train RoboCup simulation league soccer agents with that method. Reinforcement learning is a useful framework for dealing with sequential decision problems. In this unsupervised learning method the search space is explored by trial and error. Agents learn the mapping from situations to actions by getting reward signals.

It could be problematic to apply reinforcement learning to real robots due to the usually large number of training iterations.

A possibly better learning method would be *genetic algorithms* (cf. [56, 57]). With that method, a fixed set of parameters can be optimized using the methods of natural evolution. These parameters could be passed to *XABSL* options via input symbols. If starting with a manually tuned parameter set, possibly the behaviors would improve after a few generations already.

5.2.4 Detecting Strategies and Adapting to Opponent Teams

As already mentioned in the preface of this section, the capability to detect opponent strategies could improve the performance of the robots. For instance, it would be helpful for the agents to know under which conditions the opponent goalkeeper leaves its own penalty area. Visser et al. [71, 21] and Riley and Veloso [65] have shown how to build qualitative and quantitative opponent models that can be used to adapt the strategies of the own team. However, this work was done in the simulation league, where a “coach” agent has a centralized and complete view of the world.

At the moment, such a complete and accurate world model can not be obtained in the Sony Four Legged League. The robots have only a poor and inconsistent knowledge about the positions of the opponent robots. Up to now, the robots do not recognize why the ball has moved. But this basic knowledge would be necessary in order to recognize opponent strategies.

Nevertheless, it could be possible to adapt to opponent teams by measuring the success of the own actions. The easiest way to do that could be measuring the goal difference (which is sent through wireless communication by a referee program). Much more difficult would be is the observation of the success of single behaviors. As already mentioned above, up to now the perception and modelling capabilities of the robots are too limited to get reliable performance measures. But a good criterion for success could be how often a behavior was stopped. Usually, behaviors written in *XABSL* have target states. Only if a behavior was successful, such a target state is reached.

For that, the *XabslEngine* would have to be extended. For all options, the number of activations, average option and state activation times, and the number of finished activations (reached target states) would have to be measured. These data could be made available for decision making either by extending the *XABSL* language for language elements that allow to access the statistics or by passing it to the options using custom input symbols. Possibly, such statistics could be used to choose more successful behaviors more often.

5.3 Acknowledgements

The author should like to thank the members of the *Aibo Team Humboldt*, Matthias Jüngel, Uwe Düffert, Jan Hoffmann, Michael Spranger, Viviana Goetzke, Benjamin Altmeyer, Daniel Göhring, and the members of the *GermanTeam*, especially Max Risler, Dirk Thomas, Mark Dassler, Thomas Röfer, Tim Laue, Artur Cesarz, Matthias Hebbel, Carsten Schumann, Jochen Kerdels, Michael Wachter. They filled the *XABSL* framework with content, provided bug fixes, and made suggestions how to improve *XABSL*.

Special thanks go to Matthias Jüngel for critically discussing all the developments and helping in almost all the implementations, Max Risler and Matthias Jüngel for being the main contributors of behavior implementations, Michael Spranger for the implementation of the *XABSL* profiler and extensions of the *XabslEngine*, to Thomas Röfer and Uwe Düffert for technical advice, and to Almut Stracke for reading the first draft and correcting some of the errors related to the use of the English language.

Moreover, the author is in debt to the members of the artificial intelligence lab for helpful discussions, especially Hans-Dieter Burkhard, Joscha Bach, Ralf Berger, and Michael Gollin.

The Deutsche Forschungsgemeinschaft supported this work through the priority program “Cooperating teams of mobile robots in dynamic environments”.

Bibliography

[1] RoboCup website. 1997. http://www.robocup.org.

[2] German Team web site. 2002. http://www.robocup.de/germanteam.

[3] Aibo Team Humboldt web site. 2003. http://www.aiboteamhumboldt.com.

[4] Toshiaki Arai and Frieder Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems, C. Castelfranchi, W. Lewis Johnson (Eds.)*, pages 11–18, 2002. Volume 1.

[5] R. C. Arkin and D. MacKenzie. Temporal coordination of perceptual algorithms for mobile robot navigation. *IEEE Transactions on Rootics and Automation*, 10(3):276–286, 1994.

[6] Ronald C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4), 1989.

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

[8] Minoru Asada, Hiroaki Kitano, Itsuki Noda, and Manuela Veloso. RoboCup: Today and tomorrow - what we have learned. *Artificial Intelligence*, 1999.

[9] AT&T. GraphViz homepage. 2000. http://www.research.att.com/sw/tools/graphviz/.

[10] Tucker Balch. The ascii robot soccer homepage. 1995. http://www-2.cs.cmu.edu/ trb/soccer/.

[11] Ralf Berger, Michael Gollin, and H.-D. Burkhard. AT Humboldt 2003 – team description. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[12] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. W3C recommendation: Extensible markup language (XML) 1.0 (second edition). 2000. http://www.w3.org/TR/REC-xml.

[13] Rodney A. Brooks. A robust layered control system for a mobile robot. In *IEEE Journal of Robotics and Automation*, volume 2, pages 14–23, 1986.

[14] Rodney A. Brooks. The behavior language; user's guide. Technical Report AIM-1127, MIT Artificial Intelligence Lab, 1990.

[15] R. Brunn, U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, S. Petters, M. Risler, Th. Röfer, and A. Sztybryc. GermanTeam 2001. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Computer Science, pages 705–708. Springer, 2001. More detailed in: http://www.tzi.de/kogrob/papers/GermanTeam2001report.pdf.

[16] Hans-Dieter Burkhard, Joscha Bach, Ralf Berger, Birger Brunswiek, and Michael Gollin. Mental models for robot control. In *M.Beetz et al (Eds.): Advances in Plan-Based Control of Robotic Agents*, Lecture Notes in Artificial Intelligence, pages 71–88, 2002.

[17] James Clark. W3C recommendation: XSL transformations (XSLT) version 1.0. 1999. http://www.w3.org/TR/XSLT.

[18] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[19] Uwe Düffert. Vierbeiniges laufen: Modellierung und optimierung von roboterbewegungen: Diplomarbeit, 2004.

[20] Uwe Düffert, Matthias Jüngel, Tim Laue, Martin Lötzsch, Max Risler, and Thomas Röfer. GermanTeam 2002. In *RoboCup 2002 Robot Soccer World Cup VI, Gal A. Kaminka, Pedro U. Lima, Raul Rojas (Eds.)*, number 2752 in Lecture Notes in Artificial Intelligence. Springer, 2003. More detailed in http://www.tzi.de/kogrob/papers/GermanTeam2002.pdf.

[21] C. Drücker, S. Hübner, U. Visser, and H.-G. Weland. "As time goes by" - using time series based decision tree induction to analyze the behaviour of opponent players. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Artificial Intelligence, pages 325–330. Springer, 2002.

[22] F. Dylla, A. Ferrein, and G. Lakemeyer. Specifying multirobot coordination in ICPGolog – from simulation towards real robots. In *Proc. of the Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World Modeling, Planning, Learning, and Communicating (IJCAI 03)*, 2003.

[23] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[24] David C. Fallside. W3C recommendation: XML schema part 0: Primer. 2001. http://www.w3.org/TR/xmlschema-0/.

[25] Masahiro Fujita and Hiroaki Kitano. Development of an Autonomous Quadruped Robot for Robot Entertainment. *Autonomous Robots*, 5(1):7–18, 1998.

[26] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 1999.

[27] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 809–815. MIT Press, 1992.

[28] Jan Hoffmann and Daniel Göhring. Sensor-Actuator-Comparison as a Basis for Collision Detection for a Quadruped Robot. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[29] Jan Hoffmann, Matthias Jüngel, and Martin Lötzsch. A vision based system for goal-directed obstacle avoidance used in the RC 03 obstacle avoidance challenge. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[30] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.

[31] N. R. Jennings and Y. Lesperance. *Intelligent Agents VI: Agent Theories, Architectures, and Languages. IJCAI'99 Workshop (ATAL)*. Number 1757 in Lecture Notes in Artificial Intelligence. Springer, 2000.

[32] Matthias Jüngel. A vision system for robocup: Diploma thesis, 2004.

[33] Matthias Jüngel. Using layered color precision for a self-calibrating vision system. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[34] Matthias Jüngel, Jan Hoffmann, and Martin Lötzsch. A real-time auto-adjusting vision system for robotic soccer. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004.

[35] David Kinny and Michael Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, number 1193 in Lecture Notes in Artificial Intelligence. Springer, 1997.

[36] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In *Proceedings of the First International Conference on Autonomous Agents (Agents'97), W. Lewis Johnson and Barbara Hayes-Roth (Eds.)*, pages 340–347. ACM Press, 1997.

[37] Kurt Konolige. COLBERT: A language for reactive control in Sapphira. In *KI-97: Advances in Artificial Intelligence – Proceedings of the 21st Annual German Conference on Artificial Intelligence, G. Brewka, C. Habel, and B. Nebel (Eds.)*, number 1303 in Lecture Notes in Artificial Intelligence, pages 31–52. Springer, 1997.

[38] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence: JETAI*, 9(1):215–235, 1997.

[39] J. Kosecka and R. Bajcsy. Discrete event systems for autonomous mobile agents. In *Proceedings Intelligent Robotic Systems '93 Zakopane*, pages 21–31, 1993.

[40] Georgia Tech Mobile Robot Laboratory. Missionlab. User manual for missionlab version 6.0, 2003. http://www.cc.gatech.edu/aimosaic/robot-lab/research/MissionLab/mlab_manual-6.0.pdf.

[41] Tim Laue and Thomas Röfer. A behavior architecture for autonomous mobile robots based on potential fields. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[42] J. A. Leite, A. Omicini, L. Sterling, and P. Torroni. *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003. Melbourne, Victoria, July 15th, 2003. Workshop Notes*. 2003.

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

[44] Martin Lötzsch. DotML Documentation. 2003. http://www.martin-loetzsch.de/DOTML.

[45] Martin Lötzsch. XABSL web site. 2003. http://www.ki.informatik.hu-berlin.de/XABSL.

[46] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[47] A. S. Rao M. P. Singh and M. Wooldridge. *Intelligent Agents IV: Agent Theories, Architectures, and Languages. IJCAI'97 Workshop (ATAL)*. Number 1365 in Lecture Notes in Artificial Intelligence. Springer, 1998.

[48] D. MacKenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, 1997.

[49] Patti Maes. Situated agents can have goals. In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back, P. Maes (Editor)*, pages 49–70. MIT Press, 1990.

[50] Jonathan Marsh and David Orchard. W3C candidate recommendation: XML inclusions (XInclude) version 1.0. 2002. http://www.w3.org/TR/xinclude/.

[51] Marvin Minsky. *The Society of Mind*. Simon and Schuster, New York, 1986.

[52] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Series in Computer Science. WCB/McGraw-Hill, 1997.

[53] J. P. Mueller, M. P. Singh, and A. S. Rao. *Intelligent Agents V: Agent Theories, Architectures, and Languages. IJCAI'98 Workshop (ATAL)*. Number 1555 in Lecture Notes in Artificial Intelligence. Springer, 1999.

[54] Jan Murray, Oliver Obst, and Frieder Stolzenburg. Towards a logical approach for soccer agents engineering. In *RoboCup 2000: Robot Soccer World Cup IV, P. Stone, T. Balch, and G. Kraetszchmar (Eds.)*, number 2019 in Lecture Notes in Artificial Intelligence, pages 199–208. Springer, 2001.

[55] David Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[56] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, 2000.

[57] Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *R. Brooks and P. Maes (editors), Artificial Life IV*, pages 190–197, 1994.

[58] Oliver Obst. Specifying rational agents with statecharts and utility functions. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Artificial Intelligence, pages 173–182. Springer, 2002.

[59] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR91), J. Allen, R. Fikes, and E. Sandewall (Eds.)*, pages 473–484. Morgan Kaufmann, 1991.

[60] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319. AAAI Press, 1995.

[61] Thomas Röfer. An architecture for a national robocup team. In *RoboCup 2002 Robot Soccer World Cup VI, Gal A. Kaminka, Pedro U. Lima, Raul Rojas (Eds.)*, number 2752 in Lecture Notes in Artificial Intelligence, pages 417–425. Springer, 2003.

[62] Thomas Röfer, Ingo Dahm, Uwe Düffert, Jan Hoffmann, Matthias Jüngel, Martin Kallnik, Martin Lötzsch, Max Risler, Max Stelzer, and Jens Ziegler. GermanTeam 2003. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear. more detailed in http://www.robocup.de/germanteam/GT2003.pdf.

[63] Thomas Röfer and Matthias Jüngel. Vision-Based Fast and Reactive Monte-Carlo Localization. *IEEE International Conference on Robotics and Automation*, 2003.

[64] Thomas Röfer and Matthias Jüngel. Fast and robust edge-based localization in the sony four-legged robot league. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[65] Patrick Riley and Manuela Veloso. Recognizing probabilistic opponent movement models. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Artificial Intelligence. Springer, 2002.

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

[67] Peter Stone and Richard S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proc. 18th International Conf. on Machine Learning*, pages 537–544. Morgan Kaufmann, San Francisco, CA, 2001.

[68] Richard S. Sutton and Andrew G. Barto. Reinforcement learning I: Introduction. In *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[69] Milind Tambe. Implementing agent teams in dynamic multi-agent environments. *Applied Artificial Intelligence*, 12(2-3):189–210, 1997.

[70] Wiebe van der Hoek. Logical foundations of agent-based computing. In *Multi-Agent Systems and Applications, Michael Luck, Vladimir Marik, Olga Stepanjova, Robert Trappl (Eds.)*, number 2086 in Lecture Notes in Artificial Intelligence, pages 50–73. Springer, 2001.

[71] U. Visser and H.-G. Weland. Using online learning to analyze the opponents behavior. In *RoboCup 2002 Robot Soccer World Cup VI, Gal A. Kaminka, Pedro U. Lima, Raul Rojas (Eds.)*, number 2752 in Lecture Notes in Artificial Intelligence, pages 78–93. Springer, 2003.

[72] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[73] Michael Wooldrige, Jörg P. Müller, and Milind Tambe. *Intelligent Agents II: Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL)*. Number 1037 in Lecture Notes in Artificial Intelligence. Springer, 1996.