XABSL - A Pragmatic Approach to Behavior Engineering

Martin Loetzsch[1,3]
[1]Sony CSL Paris
6, rue Amyot,
F-75005 Paris, France
martin.loetzsch@csl.sony.fr

Max Risler[2]
[2]Technische Universität Darmstadt
Simulation and Systems Optimization Group
Hochschulstr. 10, D-64289 Darmstadt, Germany
risler@sim.informatik.tu-darmstadt.de

Matthias Jüngel[3]
[3]Humboldt-Universität zu Berlin
AI Lab, Unter den Linden 6
D-10099 Berlin, Germany
juengel@informatik.hu-berlin.de

***Abstract*— This paper introduces the *Extensible Agent Behavior Specification Language* (XABSL) as a pragmatic tool for engineering the behavior of autonomous agents in complex and dynamic environments. It is based on hierarchies of finite state machines (FSM) for action selection and supports the design of long-term and deliberative decision processes as well as of short-term and reactive behaviors. A platform-independent execution engine makes the language applicable on any robotic platform and together with a variety of visualization, editing and debugging tools, XABSL is a convenient and powerful system for the development of complex behaviors. The complete source code can be freely downloaded from the XABSL website (http://www.informatik.hu-berlin.de/ki/XABSL/). The language has been successfully applied on many robotic platforms, mainly in the domain of *RoboCup* robot soccer. It gave the *GermanTeam* the crucial advantage over other teams to become the 2004 and 2005 world champion in the Four-Legged League and helped the team *CoPS Stuttgart* to become third in the Middle Size League in 2004.**

I. Introduction

Engineering behaviors of (multiple) autonomous agents in complex and highly dynamic environments is still a challenging problem in robotics and Artificial Intelligence. For many years, approaches from classical symbolic and knowledge based AI [23] have been dominant in these areas of research. Generating appropriate actions or "planning" was reduced to problem solving (as for example in [14]), by that requiring symbolic representations of the world and its static and dynamic constraints as well as of the impact of actions on the environment. Despite general problems with grounding meaningful and stable representations in the agent's environment (see [24] for a review), it is a difficult task to cope with the complexity of the system by means of logic when agents have to deal with noisy sensor readings, unpredictable dynamics of the world, and uncertainty of actions. As Gat [11] remarked: "Elevator doors and oncoming trucks wait for no theorem prover."

Expressing scepticism towards traditional AI research in "block world" domains, researchers came up with the *behavior based* paradigm [7], [2]. In these biologically inspired approaches direct sensor-actuator couplings control the overall behavior of an agent. To obtain more complex behaviors, several of such behavior units or modules are combined continuously [1], competitively [19], in layers [5], or state based. Although impressive behaviors have been realized with such approaches, it still needs to be shown how to scale up these systems.

Many researchers in the field of autonomous agents try to minimize the role of the designer. Some of them propose general action selection mechanisms that "automatically" choose between different options. For example, alternative behaviors could provide an activation level based on their utility in the current state of the environment. An automated selection mechanism could choose the behavior with the highest activation. Other researchers build systems that are able to learn complex hierarchical interactions with the environment by specifying the learning problem (as for example in [3]).

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

In this paper we propose the *Extensible Agent Behavior Specification Language* (XABSL) as a pragmatic and formal approach to the design of agent behavior. Hierarchies of finite state machines make the system modular and ensure the reusability of behaviors in different contexts as well as the extensibility of implementations. Section II introduces the architecture behind XABSL, section III describes the language and the runtime system, and section IV shows how XABSL has been applied in different domains. Due to space limitations, this paper can only serve as an introduction –

technical details, a language reference, and an XABSL demo containing the complete source code can be found on the XABSL website [15].

II. Hierarchies of Finite State Machines

XABSL is a language to describe the behaviors of an agent with a set of finite state machines that are organized in a hierarchy. The current state of the whole set of state machines is defined by the current states of a subset of single state machines which can be defined as a directed path. The starting node of this path is given by the current state of the distinguished root state machine in the hierarchy. Each state machine is called an *option* and the current states of the subset of options along this path the *option activation path*. The set of options is called *option graph*.

This section describes how options are connected among each other and arranged in a hierarchy, how the option activation path is updated, and how the actions are derived from the current option activation path. The behavior of the agent is the result of a sequence of such actions. A ball grabbing behavior developed by the *GermanTeam* (cf. sect. IV) for robotic soccer with Aibo robots serves as an example.

A. How the State Machines Interact with the Environment

An XABSL behavior implementation is always a part of a more complex agent program. The surrounding software has to process the sensor readings, build up (if necessary) a world model, manage the communication with other agents, control the actuators, and so on. At some point in such a *sense-think-act cycle* (usually when new data is available from the main sensor), the program passes the control to the XABSL system to update the option activation path. To access the information about the world that is needed for decision making, symbolic representations are used. Therefore, the world model of the agent system is divided into simple, typed, and non-structured information items, called *input symbols*.

There are two ways to control the actions of the robot: *basic behaviors* and *output symbols*. *Basic behaviors* are parameterized actions that can be activated by the last state of the option activation path (a state that has no subsequent option). Note that the implementation of the basic behaviors is not a part of XABSL. Usually the main actions (like locomotion) of the agent are controlled by the basic behaviors. *Output symbols* are boolean, enumerated, or decimal values. Each single state of the option activation path can modify a subset or all of the output symbols. States closer to the end of the path can re-modify symbols that have already been modified by preceding states. Output symbols can be used to control perception processes or additional actuators.

B. How Options are Organized in a Hierarchy

An XABSL behavior consists of a set of behavior modules called *options*. Each option is a finite state machine (cf. fig. 1). In each option, exactly one state is marked as the *initial state*. An arbitrary number of states can be declared as *target states* in order to indicate that a behavior is finished. Each

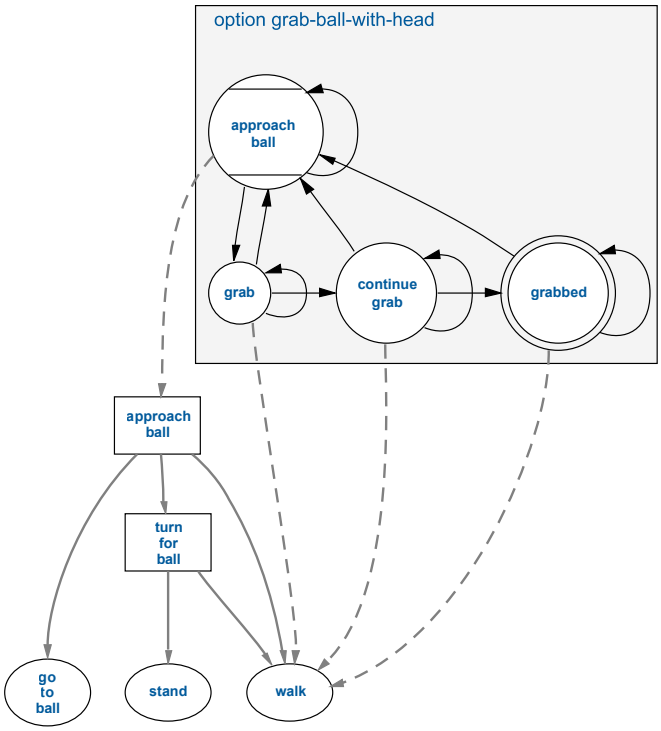

Fig. 1. An option's internal state machine. Circles denote states, the circle with the two horizontal lines denotes the initial state, the double circle denotes a target state. An edge between two states indicates that there is at least one transition from one state to the other. The dashed edges show which other option or basic behavior becomes activated when the state is active.

state of such a state machine is associated with at most one subsequent option or basic behavior. Note that more than one state can be connected to the same subsequent option or basic behavior.

This association of the states of an option with subsequent options allows to create complex behaviors that are composed from simpler ones. Thus options can use a set of other subordinated options to realize a certain behavior. For example in figure 2, the option *"handle-ball-at-opponent-border"* is composed of the option *"approach-and-turn-and-kick"* and the option *"turn-around-ball-and-kick"*.

Each option can be used from more than one other option. This allows for reusing the same behaviors in different contexts. E.g. in figure 2 the option *"approach-ball"* is used by *"grab-ball-with-head"* and *"approach-and-turn"*. This helps behavior developers to modularize their agent's behaviors. In the example, only one behavior for ball approaching was developed and fine-tuned and then used by various other different options.

The option hierarchy can be seen as a rooted directed acyclic graph, called the *option graph*. There is only one source (vertex with no incoming edges) in the option graph - the vertex that represents the *root option*. The sinks of the graph are the vertices that represent the options that have no subsequent options.

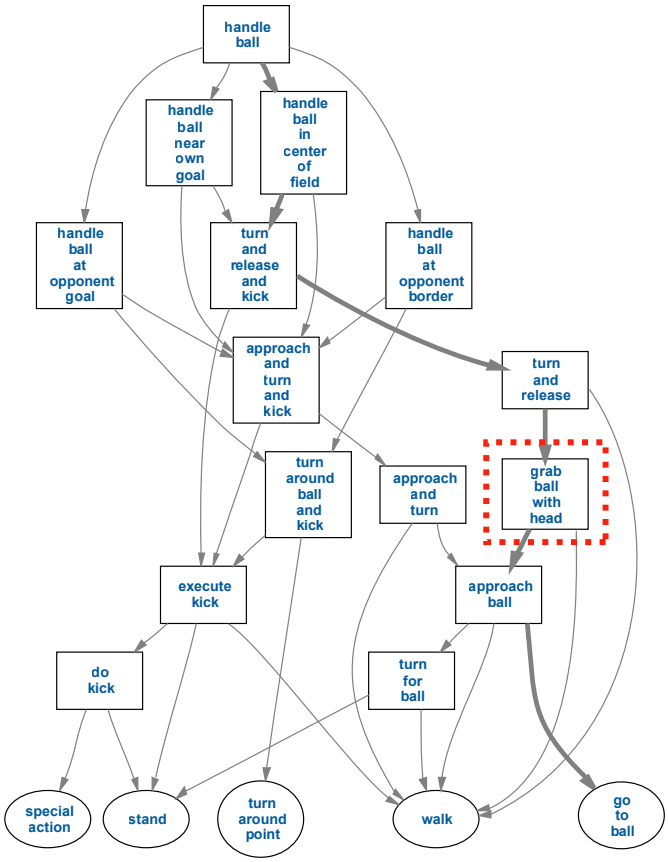

Fig. 2. An example option graph. Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from within an option. The thick edges mark one of the many possible option activation paths. The internal state machine of option *"grab-ball-with-head"* (dashed rectangle) is shown in figure 1.

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

  state grab {
```

Fig. 3. Example XABSL source code for the option *"grab-ball-with-head"*. It starts with the definition of a common decision tree (a decision tree that applies to all states of the option) and then continues with the implementation of the state *"approach-ball"*. Here the source code is shown in the editor of Microsoft Visual Studio, for which an XABSL syntax highlighting and code completion plugin exists.

C. How Options are Activated and Actions are Generated

The current state of the option graph is defined by the option activation path. The starting vertex of this path represents the current state of the root option. The last state may activate a basic behavior.

To define the possible transitions between the states each state has a *decision tree*, which selects a transition to either another or the same state. For the decisions, parameters passed by higher options, and input symbols such as the world state, other sensory information, and messages from other agents can be used. As timing is often important, the durations that the state and the option have already been active are provided. In addition, it can be queried whether the subsequent option has reached one of its target states. As each state has its own decision tree, the state transitions are not only dependent on the representation of the environment's state but also on the decisions that were made in the past. When the active state is taken into account, hysteresis functions between states are possible. Thus, behaviors can be preferred once they have been selected in order to avoid oscillations.

The update of the current option activation path of the option graph starts from the root option. The decision tree of the current state of the root option is executed to determine the new current state (which can of course be the same as before). This state is the first state in the option activation path. If the subsequent option associated with the current state was not active in the last step, the current state of this subsequent option is set to its initial state. Then the decision tree of the current state of the subsequent option is executed leading to a new current state, which is added to the option activation path. This process is repeated until the subsequent behavior of a new current state has no subsequent option. Each time a decision tree activates another or the same state, the newly activated state sets the parameters of the subsequent option or associated basic behavior and the state's output symbols.

III. The Extensible Agent Behavior Specification Language (XABSL)

This section gives a brief overview over the language XABSL, the runtime system *XabslEngine*, and some of the tools that were developed in conjunction with the language. These issues are discussed in more detail in [16] and a complete language reference and API documentation can be found at [15].

A. Behavior Specification in XABSL

Agent behaviors based on the architecture described in the previous section can be described with XABSL. Figure 3 shows an example. There is an XABSL-compiler compiler written in Ruby that can generate four different types of

documents from an XABSL document: an intermediate code for the runtime system, debug symbols to be used in debugging tools, symbol files for code completion and syntax highlighting for a variety of editors, and an XML representation XABSL specifications. The XML representation can easily be parsed by supporting tools e.g. an XSLT processor can be used to generate an extensive HTML documentation containing SVG (Scalable Vector Graphics) charts for the option graph, each option, and each state. Note that the figures 1 and 2 were generated automatically from XABSL sources.

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

Symbols are defined in XABSL instance documents to formalize the interaction with the software environment. Interaction means access to input functions and variables (e.g. from the world model) and to output functions (e.g. to set requests for other parts of the information processing). For each variable or function that one wants to use in certain conditions, a symbol has to be defined. This makes the XABSL framework independent from specific software environments and platforms. The developer may decide whether to express complex conditions in XABSL by combining different input symbols with boolean and decimal operators or by implementing the condition as an analyzer function in C++ and referencing the function via a single input symbol.

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

B. Runtime System

The class library *XabslEngine* is the XABSL runtime system. It is written in plain ANSI C++ and it is platform and application independent. To run the engine in a specific software environment, only mechanisms for file access and error handling have to be adapted to the target platform. The engine parses and executes the intermediate code that was generated from XABSL documents. It links the symbols from the XABSL specification that are used in the options and states to the variables and functions of the agent platform. Therefore, for each used symbol an entity in the software environment is registered to the engine. Basic behaviors are written in C++ and also registered to the engine at startup. The class library provides extensive debugging interfaces for monitoring and manipulating nearly all internal states of the engine. A complete API documentation is available at the XABSL web site [15].

Based on the engine's debugging interfaces it is easy to develop a tool which can display the option activation path, the parameters and execution times of options, states, and basic behaviors, as well as the values of input and output symbols. Vice versa, single options or basic behaviors can be selected and parameterized manually for execution. Figure 4 shows such a tool. Additionally, the *Xabsl Profiler* can be used to analyze behaviors over time. For that, log files containing the option activation path are recorded and visualized in such a way as to show the length of time states and options were active. This helps to detect state oscillations or unused states.

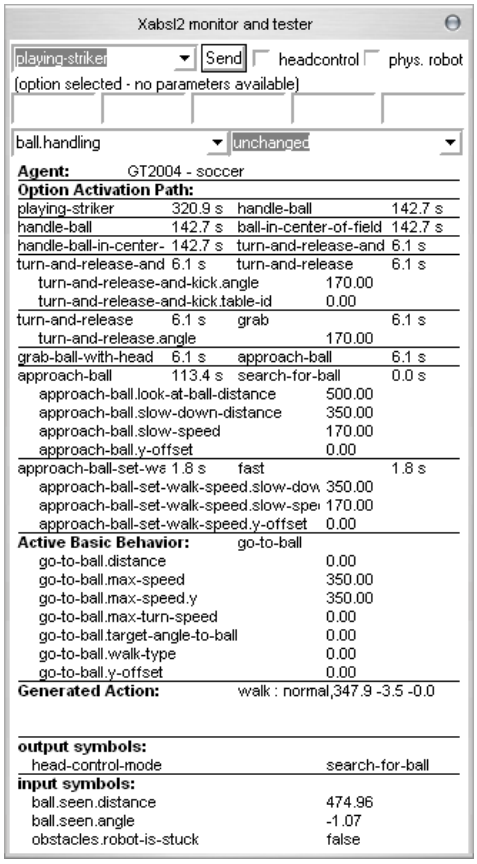

Fig. 4. An example for a XABSL monitoring tool using the debugging interfaces of the *XabslEngine* (inside the GermanTeam's *RobotControl* application.

C. Discussion

The main difference between XABSL and other behavior programming and planning languages as for example the *Behavior Language* [6], *COLBERT* [13], the *Configuration Description Language (CDL)* [18], or *PDDL* [20] is the way how it is integrated into the target platform. XABSL is much more lightweight than these as it does not impose any constraints on the agent architecture or the software design of the robotic system. Instead, programmers can easily replace their existing planning and control programs by the XabslEngine run-time system and start implementing their behaviors in XABSL.

The fact that XABSL does not model a complete agent system including sensing and acting but only provides an action selection mechanism means that the XABSL system can not be exclusively labeled as reactive or deliberative. It is

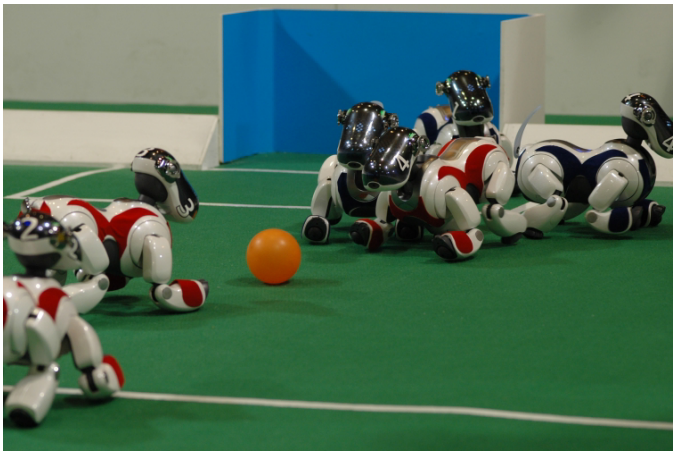

Fig. 5. A soccer game in the *RoboCup Four-Legged League*.

possible to design completely reactive agents that do not have a persistent world model and it is also possible to use complex symbolic world models as an input to a highly deliberative XABSL agent.

XABSL is not in opposition to the approaches mentioned in the introduction. It is possible (and has often been done) to use behavior-based techniques in basic behaviors, to learn parameters of options or basic behaviors, to learn conditions for state transitions, to coordinate multiple agents, or to use abstract planning algorithms and provide the results to XABSL options by input symbols.

It is the choice of hierarchical FSM that makes XABSL more scalable and easier to extend. Adding an option to an XABSL behavior specification never has side-effects on existing behaviors. Once a new behavior (both a composite option and an atomic basic behavior) has been tested and fine-tuned, it can be easily integrated in different other options, without being dependent on the different contexts of these behaviors. This is because in each of these options the decision when to activate the new subordinated behavior only depends on their state and purpose.

The next section shows how XABSL was applied in various different agent architectures.

IV. Applications

So far, XABSL is mostly applied in the *RoboCup* [12] robot soccer domain, a common testbed and benchmark problem for research in many fields of artificial intelligence and robotics. First versions of the system [17] were developed in 2001 by the *GermanTeam* [22], a group of several German researchers competing in the *RoboCup Four-Legged League* (cf. fig. 5). In this league, teams of four Sony's four legged Aibo robots [9] play soccer against each other. The main characteristic of this league is the complexity of physical actions that have to be employed both for interaction and perception. As the opening angle of the 208×160 pixels camera is only 45 degrees wide and thus the robot only perceives small portions of the field, the obtained world model is very unreliable and noisy. Additionally, walking and ball handling with four legs results in high uncertainty of actions.

Fig. 6. Team *CoPS* in the *RoboCup Middle Size League*.

The GermanTeam developed a rich set of basic behaviors for obstacle avoidance, navigation, and ball handling. Based on that, more and more complex behaviors were composed from simpler ones. In general, the lower behaviors in the option hierarchy such as ball handling or navigation tend to be more short-term and reactive as they have to react instantly on changes in the environment. The more high-level behaviors such as waiting for a pass, positioning, or role changes try to avoid frequent state changes and make more deliberative and long-term decisions. A successful behavior in the Four-Legged League usually consists of about 50 - 80 options. An example can be found at [15].

Another domain of application is the *RoboCup Middle Size League* (cf. fig. 6). In that league, custom-made wheel-based robots are usually equipped with omni-vision cameras and laser range finders and therefore have rather precise world models. For example the team *Cooperative Soccer Playing Robots Stuttgart (CoPS)* [8] easily encapsulated their existing behaviors for navigation and dribbling in XABSL basic behaviors and used the language itself mainly for very high level behaviors such as role assignments or game flow. Additionally, they developed a Petri Net based modelling tool that generates XABSL source code for specifying cooperation between robots.

In parallel to AI and robotics research and without much reciprocal recognition, the computer game community faces similar problems with similar approaches when designing the behavior of virtual creatures [10], [21]. Since 2004, several game programmers started using XABSL for their developments.

To support behavior engineers when employing XABSL on their own agent platform, an example XABSL behavior implementation was made for the ASCII Soccer simulator [4]. In this very simple soccer simulation the field, two teams of four players each, and the ball are displayed on a text terminal (cf. fig. 7). The players are able to access a nearly complete world model and the action set of the agents is very limited: They can either move to one of the eight neighboring places or kick. The simplicity of this environment made it possible to develop a competitive XABSL example agent team with dynamic role assignments, supporter positioning, passing, and

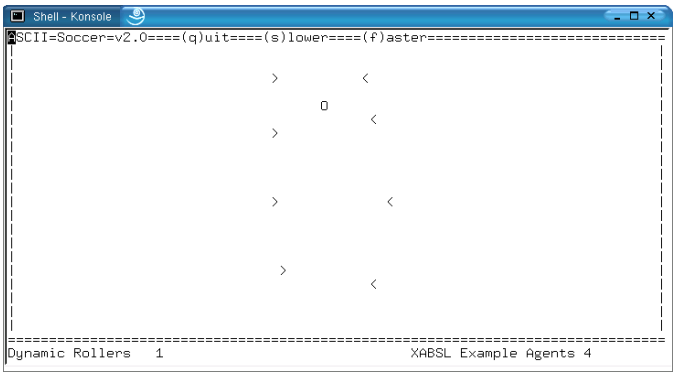

Fig. 7. A scene from an ASCII Soccer game.

dribbling in a short time. This implementation also shows that the XABSL language, the tools and the executing engine are really independent from the developments made for the robot soccer environments.

The ASCII Soccer XABSL example implementation can be downloaded together with the complete source code and tools from the XABSL web site [15].

V. Conclusions

This paper introduced the *Extensible Agent Behavior Specification Language* for the convenient developing of the behavior of autonomous agents. State based techniques allow for dealing with uncertainty in highly dynamic environments. Composing state machine based options in hierarchies makes behaviors reusable in different contexts and by that enables behavior designers to develop scalable and complex behaviors.

Although XABSL was initially developed for robotic soccer, it is not a soccer programming language – there are no language elements of concepts that are specific for soccer applications. The language and the run-time system *XabslEngine* are application and platform independent and can be relatively easily employed in any agent system.

The modular nature of XABSL supports the development of behaviors in a team (for example more than 20 team members of the *GermanTeam* were involved in the developing and tuning of their behaviors). New options can be easily added to existing ones without having negative side effects. With the debugging interfaces of the *XabslEngine* new options can be tested separately before they are used by higher-level options. Improved versions of existing options can be developed in parallel and are easy to compare with previous ones. A constantly extending library of well tuned low-level behaviors can be reused in different contexts for the creation of new options.

XABSL becomes increasingly wide spread. By now, it is used by more than 25% of the teams in the RoboCup Four-Legged League, but it is also applied on other robots in the RoboCup Middle Size and Humanoid League. It helped the *GermanTeam* to become the 2004 and 2005 world champion in the Four-Legged League. Although this success was of course also based on many other achievements, we believe that the ability of the team to develop an adopt very complex and efficient behaviors – even during the ongoing competition – played a key role in winning these titles.

Acknowledgment

The authors would like to thank all persons who contributed to the success of the XABSL system. We thank the members of the GermanTeam, especially Bastian Schmitz for writing the new XABSL compiler, Michael Spranger, Uwe Düffert, and Thomas Röfer. Parts of this research were funded by Deutsche Forschungsgemeinschaft (DFG), Schwerpunktprogramm 1125 and the EU FET ECAgents Project IST-1940.

References

[1] R. C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4), 1989.
[2] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
[3] B. Bakker and J. Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of IAS-8*, pages 438–445, 2004.
[4] T. Balch. The ascii robot soccer homepage, 1995. http://www-2.cs.cmu.edu/~trb/soccer/.
[5] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
[6] R. A. Brooks. The behavior language; user's guide. Technical Report AIM-1127, MIT Artificial Intelligence Lab, 1990.
[7] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
[8] T. Buchheim et al. Team description paper 2004 CoPS stuttgart. In *RoboCup 2004: Robot Soccer World Cup VIII*, LNAI. Springer, 2005.
[9] M. Fujita and H. Kitano. Development of an autonomous quadruped robot for robot entertainment. *Autonomous Robots*, 5(1):7–18, 1998.
[10] J. D. Funge. *Artificial Intelligence For Computer Games: An Introduction*. AK Peters, Ltd, 2004.
[11] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings AAAI-92*, pages 809–815. MIT Press, 1992.
[12] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The robot world cup initiative. In *Proceedings of Agents'97*, pages 340–347. ACM Press, 1997.
[13] K. Konolige. COLBERT: A language for reactive control in Sapphira. In *KI-97: Advances in Artificial Intelligence*, number 1303 in LNAI, pages 31–52. Springer, 1997.
[14] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
[15] M. Loetzsch, M. Risler, and M. Jüngel. XABSL web site, 2006. http://www.informatik.hu-berlin.de/ki/XABSL.
[16] M. Lötzsch. XABSL - a behavior engineering system for autonomous agents. Diploma thesis. Humboldt-Universität zu Berlin, 2004. Available online: http://www.martin-loetzsch.de/papers/diploma-thesis.pdf.
[17] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *LNAI*, pages 114–124. Springer, 2004.
[18] D. MacKenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, 1997.
[19] P. Maes. Situated agents can have goals. In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 49–70. MIT Press, 1990.
[20] D. McDermott. Pddl — the planning domain definition language, 1998.
[21] S. Rabin, editor. *AI Game Programming Wisdom*, volume 2. Charles River Media, Inc., 2002.
[22] T. Röfer et al. Germanteam 2005. In *RoboCup 2005: Robot Soccer World Cup IX*, LNAI. Springer, 2006. To appear.
[23] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, 1995.
[24] T. Ziemke. Rethinking grounding. In *Understanding Representation in the Cognitive Sciences – Does Representation Need Reality?*, pages 177–190. Plenum Press, 1999.