Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL*

Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel

Institut für Informatik, LFG Künstliche Intelligenz, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
{loetzsch,bach,hdb,juengel}@informatik.hu-berlin.de

Abstract. Specific behavior description languages prove to be suitable replacements to native programming language like C++ when the number and complexity of behavior patterns of an agent increases. The XML based *Extensible Agent Behavior Specification Language* (*XABSL*) also simplifies the process of specifying complex behaviors and supports the design of both very reactive and long term oriented behaviors. XABSL uses hierarchies of *behavior modules* called *options* that contain state machines for decision making. In this paper we introduce the architecture behind *XABSL*, the formalization of that architecture in XML and the software library *XabslEngine* that runs the formalized behavior on an agent platform. The *GermanTeam* [9] employed *XABSL* in the RoboCup *Sony Four Legged League* competitions in Fukuoka.

1 Introduction

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. The Sony robots have four legs with 3 DOF each, and a head with 3 DOF. Instead of kicking with a single kicking device like in the middle or small sized league, this allows for a lot of different kicking skills using legs, body, or even head, which often require preparatory movements. Instead of moving on wheels many different styles of walking are used in different situations. 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 behavior.

For perception, the Sony robots need a set of perception behaviors, too. Because the field of vision, the image quality and the quality of the other sensors are very limited, information has to be collected over time, the movement of legs and head has to be coordinated with current vision needs and the perception process needs to be supported by methods like active vision. Usually, the related behaviors have to be merged with the other movement skills.

This huge set of abilities results in the need for a complex behavior control architecture that integrates many behavior patterns. It should be modular in

* The Deutsche Forschungsgemeinschaft supports this work through the priority program "Cooperating teams of mobile robots in dynamic environments".

D. Polani et al. (Eds.): RoboCup 2003, LNAI 3020, pp. 114–124, 2004.

the meaning that behavior patterns can be reused in different contexts. It has to support reactive and realtime decision making as well as long term deliberative behaviors. The set of behaviors needs to be easy to extend - adding new behaviors should not have side effects on other ones.

We found C++ not well suited for specifying agent behaviors. Especially extension and maintenance of complex behavior control systems may become a tedious and error prone task. More high level behavior specification languages allow for a separation of the behavior design from the implementation of the agent platform.

1.1 Related Work

In all RoboCup leagues, intentional cooperation and the pursuit of long term strategic behavior remain a challenge. According to the dynamics of soccer, the agents act with only very limited foresight.

Most teams in RoboCup are using **layered architectures**, with comparatively reactive behaviors (basic skills) at the lowest level (cf. e.g. [2, 15, 8]). Ordering different behaviors on layers allows to follow different goals in parallel. Behaviors on higher levels invoke or activate behaviors on lower levels. As long as the architecture has to manage only few basic behaviors, the separation of behaviors in two or three layers may be sufficient. But in our experience, it becomes very difficult to control more than a few basic behaviors without introducting further hierarchies, when their usage depends on a careful analysis of the situation, when they require complex preconditions to be achieved and when their performance needs a considerable amount of time.

There are other attempts to use **behavior languages** in order to simplify the process of behavior development. For example, GOLOG [12] is an logic based robot control language. Funge [11] developed the *cognitive modeling language (CML)* for the domain of computer games. Obst and Stolzenburg [16, 1] employ UML state charts for specifying multiagent systems. They follow a layered state machine approach with a fixed number of layers. They used UML because there exists a rich set of easy-to-adapt editors for editing state charts.

State machines are well suited for behavior modelling (cf. e.g. [16, 1, 7]). The decision which action is executed next depends not only on the environment but also on the last state. That allows to keep behaviors stable and to define hystereses between two behaviors for avoiding oscillations when the sensor readings are noisy.

1.2 Main Contributions

In this paper we present a flexible, open hierarchical behavior control architecture. It consists of state machines which manage the transitions to new behaviors according to the last state and the recent situation. In a flat architecture, the number of transitions between states increases very fast with the number of states. Therefore we use options to encapsulate a limited number of states and

transitions according to their abstractness. The options form a rooted directed acyclic graph. In section 2.1 we describe that approach in detail.

Based on that architecture, we introduce *XABSL* as an XML based behavior language that allows to completely specify the behavior of autonomous agents. The development of a robot control includes the design of a hierarchy of options and the implementation of their internal state machines. *XABSL* supports both tasks using the advantages of XML technologies. The *XABSL* framework contains a variety of visualization and debugging tools. The runtime system *XabslEngine* (section 2.3) executes the behaviors written in XABSL.

The *GermanTeam* [9] competes in the Sony Four Legged League and is a national team that consists of separate teams from five German universities, amongst them the Humboldt University in Berlin. Section 3 shows how *XABSL* was employed by that team and which experiences were made in the competitions. *XABSL* could be proven to allow the efficient integration of program parts from different groups. It is possible to develop a new robot control with about 50 different behaviors in only two weeks.

2 Developing Agent Behavior with XABSL

2.1 The Architecture behind XABSL

In the presented architecture an *agent* consists of a number of *behavior modules* called *options*. The options are ordered in a rooted directed acyclic graph, the *option graph* (cf. Fig. 1a), which may be expanded into a tree. The terminal nodes of that graph are called *basic behaviors*. They generate the actions of the agent and are associated with basic skills.

The task of the option graph is to activate and parameterize one of the basic behaviors, which is then executed. Beginning from the root option, each active option has to activate and parameterize another option on a lower level in the graph or a basic behavior. Within options, the activation of behaviors on lower levels is done by state machines (cf. Fig. 1b). Each state has a subsequent option or a subordinated basic behavior. Note that there can be several states that have the same subsequent option or basic behavior.

Each state has a decision tree (cf. Fig. 2) with transitions to other states at the leaves. For the decisions the agent's world state, other sensory information and messages from other agents can be used. As timing is often important, the time how long the state is already active and the time how long the option is already active can be taken into account. Additionally, each state can set special requests, that influence the information processing or determine how and where the robot should point its camera.

2.2 Behavior Specification in XML

In previous RoboCup participations the *GermanTeam* made the experience that implementing such an architecture totally in C++ is error prone and not very comfortable. The source code became very large and it was quite hard to extend

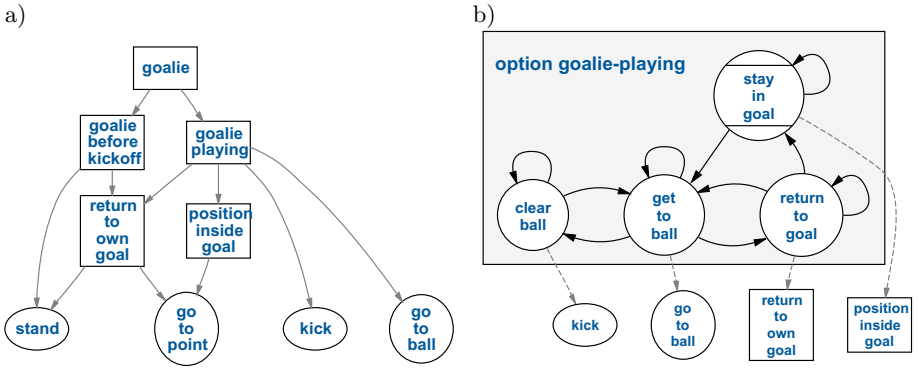


Fig. 1. a) The option graph of a very simplified goalie (this is only a simple example – the option graph developed by the *GermanTeam* for the competitions in Fukuoka contains about 50 options). Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from within an option. b) The internal state machine of the option "goalie-playing". Circles denote states, the circle with the two horizontal lines denotes the initial 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 charts were generated automatically from the XML source in Fig. 3.

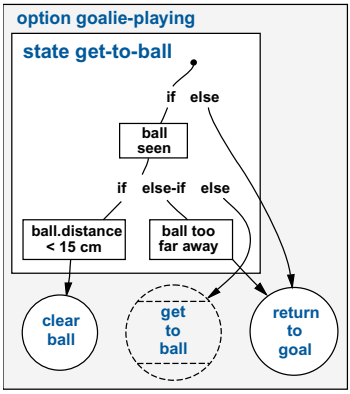
the behaviors. Therefore the *Extensible Agent Behavior Specification Language* (*XABSL*) was developed to simplify the process of specifying behaviors.

XABSL is an *XML 1.0* [4] dialect specified in *XML Schema* [10]. The reasons to use XML technologies instead of defining a new grammar from scratch were the big variety and quality of existing editing, validation and processing tools (many XML Editors are able to check if an *XABSL* document is valid at runtime), the possibility of easy transformation from and to other languages as well as the general flexibility of data represented in XML languages. Behaviors specified in *XABSL* can be easily visualized using XSLT [6] and DotML [13]. Note that the figures 1 and 2 were generated automatically from the XML source in Fig 3.

Agents based on the architecture introduced in the previous section can be completely described in *XABSL*. We have implemented language elements for options, their states, and their decision trees. Boolean logic ($||$, $\&\&$, $!$, $==$, $!=$, $<$, $<=$, $>$ and $>=$) and simple arithmetic operators ($+$, $-$, $*$,/ and $\%$) can be used for conditional expressions. 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 state) and to output functions (e. g. to set requests for other parts of the information processing). For each variable or func-

a)



b)

```
if (ball.time-since-last-seen < 2000)      //ball seen
{
  if  (ball.distance < 150)           //ball kickable
  {
    transition-to-state (clear-ball);
  }
  else if  (ball.distance > 900)  //ball too far away
  {
    transition-to-state (return-to-goal);
  }
  else
  {
    transition-to-own-state (get-to-ball);
  }
}
else
{
 transition-to-state (return-to-goal);
}
```

Fig. 2. The decision tree of the state "get-to-ball". a) Graphical notation: The leaves of the tree are transitions to other states. The dashed circle denotes a transition to the own state. b) Pseudo code of the decision tree. Note that both charts were generated automatically from the XML source in Fig. 3.

tion that shall be used for 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 instead of passing/kicking"/>
<boolean-input-symbol name="goalie-should-jump-right"
  description="A ball is right ahead and rolls into to own goal"/>
```

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

An example:

```
<if>
  <condition description="behind the ball">
    <less-than>
      <decimal-input-symbol-ref ref="self.x"/>
      <plus>
        <decimal-input-symbol-ref ref="ball.x"/>
        <decimal-value value="200"/>
      <plus>
```

```
    </less-than>
  </condition
  <transition-to-state ref="foo"/>
</if>
<else>
  <if>
    <condition description="clear right">
      <boolean-input-symbol-ref="goalie-should-jump-right"/>
    </condition>
    <transition-to-state ref="clear-right"/>
  </if>
  <else> ... </else>
</else>
```

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.

An *XABSL* behavior specification can be distributed over many files. The *GermanTeam* uses different XML files for symbol definitions, basic behavior definitions, predefined conditions, agents and options. This helps larger teams of behavior developers to work in parallel. It is easier to keep an overview over the whole agent and a version control system like CVS can be easily used.

We developed tools for generating three different types of documents from an *XABSL* instance document set:

- An *Intermediate Code* which is executed by the *XabslEngine* (*see* sction 2.3). This was done because on many embedded computing platforms (like Sony's AIBO), XML parsers are not available due to resource and portability constraints.
- *Debug Symbols* containing the names for all options, states, basic behaviors and symbols make it possible to implement platform and application dependent debugging tools for monitoring option and state activations as well as input and output symbols.
- An extensive auto-generated *HTML-documentation* containing SVG-charts for each agent, option and state which helps the developers to understand what their behaviors do.

Fig. 3 shows an example for an *XABSL* source file. For more details about the language, the *XABSL* web site [14] contains a complete language reference, the XML schema files and examples.

```
<option name="goalie-playing" initial-state="stay-in-goal"
                    description="goalie playing behavior">
  ...
  <state name="get-to-ball">
    <following-basic-behavior ref="go-to-ball"/>
    <set-output-symbol ref="head-control-mode"
                       value="search-for-ball"/>
    <decision-tree>
      <if>
        <condition description="ball seen">
          <less-than>
            <decimal-input-symbol-ref
                      ref="ball.time-since-last-seen"/>
            <decimal-value value="2000"/>
          </less-than>
        </condition>
        <if>
          <condition description="ball kickable">
            <less-than>
              <decimal-input-symbol-ref
                                  ref="ball.distance"/>
              <decimal-value value="150"/>
            </less-than>
          </condition>
            <transition-to-state ref="clear-ball"/>
        </if>
        <else-if>
          <condition description="ball too far away">
            <greater-than>
              <decimal-input-symbol-ref
                                  ref="ball.distance"/>
              <decimal-value value="900">
            </greater-than>
          </condition>
          <transition-to-state ref="return-to-goal"/>
        </else-if>
        <else>
          <transition-to-state ref="get-to-ball"/>
        </else>
      </if>
      <else>
        <transition-to-state ref="return-to-goal"/>
      </else>
    </decision-tree>
  </state>
  ...
</option>
```

Fig. 3. An example for an *XABSL* XML notation: a source code fragment for the state *get-to-ball* (cf. Fig. 2) of option *goalie-playing* (cf. Fig. 1).

2.3 The Runtime System XabslEngine

For running the compiled behavior on a target agent platform, the runtime environment *XabslEngine* has been developed. The engine is meant to be platform and application independent and can be easily employed on other robotic platforms. This results in a variety of abstract helper classes that have to be adapted to the current software environment.

The *XabslEngine* parses and executes the intermediate code. It links the symbols from the XML specification that were 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.

The following example connects the C++ variable *worldState.ballPosition.x* to the *XABSL* symbol *"ball.x"*:

```
myEngine.registerDecimalInputSymbol("ball.x",
                                    &worldState.ballPosition.x);
```

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 registered to the engine.

The engine provides extensive debugging interfaces to be able to monitor the option and state activations, the values of the symbols and the parameters

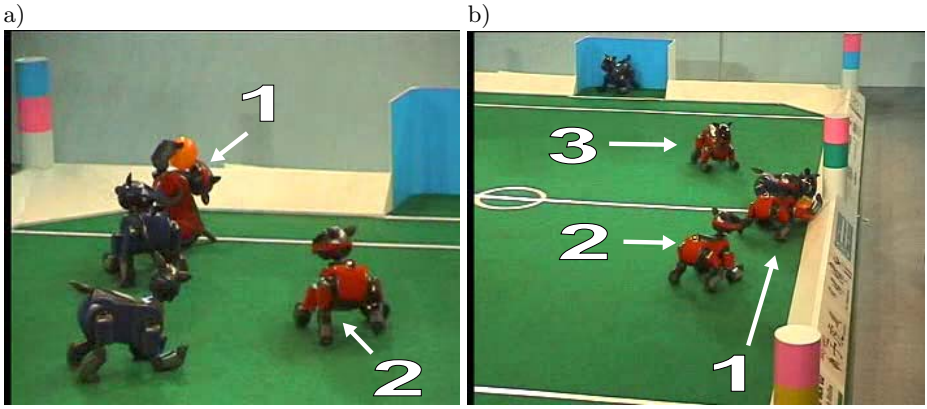


Fig. 4. Scenes from a video of a round robin game against the team *Georgia Tech* (4:1) in Fukuoka: a) *Use of communication.* The first forward (1) performs a bicycle kick directed to the opponent goal. The second forward (2) was told to wait in front of the opponent goal to be able to help if the kick fails. b) *Positioning.* The second forward (1) tries to dribble the ball into the opponent half. The defender (2) stays behind it to support the forward. The first forward (3) waits in the opponent half for a pass.

of options and basic behaviors. Instead of executing the engine from the root option, single options or basic behaviors can be tested separately.

A complete documentation of the engine, along with the code, can be found at the *XABSL* web site [14].

3 Application

XABSL was developed for the participation of the *Aibo Team Humboldt* from the Humboldt-Universität zu Berlin at the *GermanOpen* 2002 in Paderborn. Later on this approach was chosen for the participation of the *GermanTeam* in the RoboCup 2002 in Fukuoka [9]. In the competitions the *GermanTeam* won all its games except against the later finalists from CMU and UNSW.

The strength of the team was based on a big set of different behavior patterns. For instance the players employed over 16 different kicks in different situations. Amongst them the *bicycle kick* is a good method for getting the ball behind the player without previously turning around the ball. (cf. Fig. 4a) All these kicks require different behaviors for approaching the ball. Some work better for bigger ball distances, some require to grab the ball with the both front legs. Varying ball handling behaviors were chosen depending on whether the ball was in the opponent half, in the own half, at the left border, at the right border or in front of the opponent goal. *XABSL* proved to be suitable for implementing and integrating all these different abilities.

On higher levels, a set of team strategies based on communication was implemented. As it is often disadvantageous when two players try to obtain the ball the robots negotiated which of them handles the ball and which stays behind or

waits for a pass (cf. Fig. 4a). The state based architecture of *XABSL* simplifies the developing of such strategies. Each robot sends its option and state activations to all other robots so that all players know what the others plan to do. However, since the wireless communication is not always reliable, all strategies have to be able to resort to non-communicative behavior, when necessary.

Complex positioning strategies were also employed. Each player had to care for an area of responsibility which changed depending on the score, the number of own players and the distribution of opponent players on the field (cf. Fig. 4b).

Although *XABSL* is a state based architecture, continuous approaches can easily be integrated into the behaviors. A *potential field* was employed to determine an optimal dribbling direction. This direction was made available to the options by an input symbol. A *Fuzzy Logic* based basic behavior for approaching the ball was implemented. Several options used continuous *utility models* for state transitions.

The hierarchical constitution of *XABSL* allows it to make many both very short-term and reactive decisions and more deliberative and long-term decisions co-instantaneous. The lower behaviors in the option hierarchy that are responsible for ball handling react instantly on changes in the environment. The more high-level behaviors like waiting for a pass, positioning or role changes try to prevent frequent state changes to avoid oscillations.

Altogether the *GermanTeam* implemented over 50 different *options* for the games in Fukuoka. About 10 team members were involved in developing and tuning the behaviors. The modular approach of *XABSL* made it easy to extend or advance the behaviors. New options could easily be added to existing ones without having negative side effects. Better solutions of existing options could be developed in parallel and were easily to compare with the previous ones.

Additionally, to help behavior control developers who want to employ *XABSL* on their own robotic platform, an example agent was implemented for the *Ascii Robot Soccer* environment [3]. In this simple soccer simulation by Tucker Balch the field is displayed in a 78 characters long and 21 lines wide text terminal. A team of four "$>$" players plays against a team of four "$<$" players with an "o" as the ball. All agents retrieve the full information about the world and the set of possible actions is very limited. This makes the implemented *XABSL* agent simple and easy to understand. The example implementation containing the *XabslEngine* and the visualization tools can also be downloaded from the *XABSL* web site [14].

4 Conclusion and Outlook

In this paper we present an approach for behavior design for teams of autonomous agents based on hierarchical state machines. The *Extensible Agent Behavior Specification Language (XABSL)* is an XML dialect that allows to conveniently develop behaviors using that architecture. We show how the *GermanTeam* employed that language to develop complex team behaviors for the RoboCup competitions in the Sony Four Legged League. 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 code library is open source and publicly available at our website [14].

Future Work. Current developments of our behavior architecture aim at supporting the pre-deliberation of long-term strategies, which has to take place in parallel with the real-time execution of these strategies. This is done by adopting the *Double Pass architecture* [5], which has been developed for the simulation league team *AT Humboldt* of our work group. The *Double Pass architecture* annotates option hierarchies in its deliberation pass as intended, desirable or inapplicable, and executes the resulting plans in its second pass using a least commitment approach. The accommodation of additional condition types and different run-time requirements ask for extensions of *XABSL* as well as for the *XabslEngine*.

Acknowledgments

The authors would like to thank the members of the *GermanTeam* (especially Uwe Düffert, Jan Hoffmann and Max Risler) for filling the *XABSL* framework with content and Thomas Röfer for technical advice.

References

1. T. Arai and F. 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.
2. R. C. Arkin. *Behavior-Based Robotics*. The MIT Press, 1998.
3. T. Balch. The ascii robot soccer homepage. 1995. http://www-2.cs.cmu.edu/~trb/soccer/.
4. T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. W3C recommendation: Extensible markup language (XML) 1.0 (second edition). 2000. http://www.w3.org/TR/REC-xml.
5. H.-D. Burkhard, J. Bach, R. Berger, B. Brunswiek, and M. 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.
6. J. Clark. W3C recommendation: XSL transformations (XSLT) version 1.0. 1999. http://www.w3.org/TR/xslt.
7. Z. Crisman, E. Curre, C. Kwok, L. Meyers, N. Ratliff, L. Tsybert, and D. Fox. Team description: UW huskies-02. In *RoboCup 2002 Robot Soccer World Cup VI, Gal A. Kaminka, Pedro U. Lima, Raul Rojas (Eds.)*, Lecture Notes in Computer Science, 2003. to appear.
8. A. Dahlströhm, F. Heintz, M. Jacobsson, J. Thapper, and M. Öberg. The NOAI team description. 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 412–416, 2001.

9. U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, M. Risler, and T. Röfer. GermanTeam 2002. In *RoboCup 2002 Robot Soccer World Cup VI, G. Kaminka, P. Lima, R. Rojas (Eds.)*, Lecture Notes in Computer Science, 2003. to appear. more detailed in http://www.tzi.de/kogrob/papers/GermanTeam2002.pdf.
10. D. C. Fallside. W3C recommendation: XML schema part 0: Primer. 2001. http://www.w3.org/TR/xmlschema-0/.
11. J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Siggraph 1999, Computer Graphics Proceedings, Alyn Rockwood (Editor)*, pages 29–38. Addison Wesley Longman, Los Angeles, 1999.
12. 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.
13. M. Lötzsch. DotML Documentation. 2003. http://www.martin-loetzsch.de/DOTML.
14. M. Lötzsch. XABSL web site. 2003. http://www.ki.informatik.hu-berlin.de/XABSL.
15. R. R. Murphy. *An Introduction to AI Robotics*. The MIT Press, 2000.
16. O. 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, 2002.