Preprint of a paper which appeared in the proceedings of

"AAMAS08:Workshop on Formal models and methods for multi-robot systems" (Estoril, Portugal, May 13 2008)

Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL *

Max Risler and Oskar von Stryk
Simulation, Systems Optimization, and Robotics Group
Technische Universität Darmstadt
Hochschulstr.10, D-64289 Darmstadt, Germany
{risler,stryk}@sim.tu-darmstadt.de

ABSTRACT

This paper presents the latest developments of the Extensible Agent Behavior Specification Language (XABSL), a modular and scalable tool for engineering complex multi-agent behavior. It is based on hierarchical finite state machines. By the new extensions the development of cooperative multi agent behavior is supported through language elements which allow to conveniently specify how the state machines of multiple agents interact. Basic properties of XABSL are illustrated in direct comparison with Petri Net Plans and the COLBERT language using examples of basic robot behavior. More complex examples from robot soccer are used to illustrate the new extensions of XABSL. The complete system is available online on the XABSL website (http://www.xabsl.de).

1. INTRODUCTION

In this paper complex behaviors for cooperative multi-agent applications are investigated which pose a challenging task in highly dynamic environments as they are encountered in many real-world applications. Pragmatic methods are required for programming agent behaviors that are able to cope with necessary real-time requirements, only partial or noisy observability of the environment, and the unpredictability of dynamic environments [8]. A formal method for modeling and implementing cooperative multi-agent behavior is presented in XABSL, an agent programming language which is based on hierarchical finite state machines.

The next section describes related existing agent behavior engineering approaches. Section 3 describes the *Extensible Agent Behavior Specification Language (XABSL)* including recent extensions to support multi-agent cooperations. Section 4 shows some applications. Section 5 gives concluding remarks.

*Parts of this research have been supported by the German Research Foundation (DFG) within the research training group 1362 Cooperative, Adaptive, and Responsive Monitoring in Mixed Mode Environments and the special priority program 1125 on Cooperating Teams of Mobile Robots in Dynamic Environments.

AAMAS 2008 Workshop on Formal Models and Methods for Multi-Robot Systems, May 13 2008, Estoril, Portugal.

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

Figure 1: An example of a COLBERT procedure (taken from [12])

2. EXISTING APPROACHES

Machine learning approaches have been shown to be effective even for large and complex problems, as for instance, by using hierarchical reinforcement learning [1]. Nevertheless, especially when dealing with dynamic environments, developers might want to specify explicitly what actions an agent should select in certain situations. When using a machine learning approach, such explicit directives can often only be incorporated, by adapting reward functions or by modifying the learning problem. Because of these difficulties, in many real-world autonomous robot applications such approaches prove to be inappropriate and, instead, agent behaviors are programmed manually in standard programming languages.

Agent behaviors can be specified more efficiently by using formal specification methods for behavior programming such as *Behavior Language* [4], the *Configuration Description Language (CDL)* [17], and *PDDL* [18]. But some of these architectures are integrated very deeply in the software architecture of a robot system. Therefore, integration into an existing robot control software might prove to be difficult.

COLBERT is a language for reactive behavior control based on finite state machines [12]. State machines are implicitly defined via procedure specifications. The architecture facilitates hierarchical and concurrent execution of state machines. Figures 1 and 2 show an example behavior in COLBERT and how the same behavior could be addressed using hierarchical state machines as described in this paper.

A formal approach using Petri nets for modeling robot

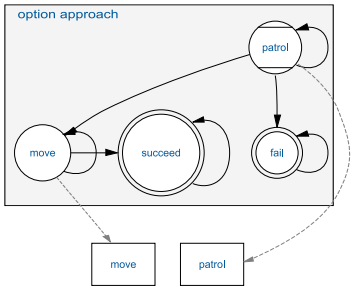


Figure 2: A possible translation of the COLBERT example to XABSL

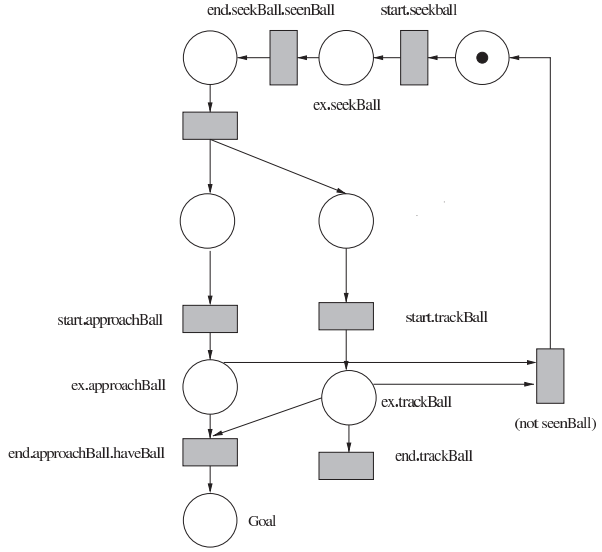


Figure 3: An example behavior modeled with Petri nets (taken from [22])

behavior can be found in [22]. Common features of this approach and the architecture presented in this paper are, hierarchical decomposition of complex behaviors, concurrent execution of partial behaviors, and support for multi-robot cooperation. Modeling behavior with hierarchical finite state machines has a similar expressiveness while in our opinion it is more intuitive by utilizing more compact behavior descriptions. An advantage of the Petri nets formalism is the possibility of the analysis and verification of certain properties of the specified behaviors. This has not yet been done using hierarchical state machines. Figures 3 and 4 give an exemplified comparison of Petri nets and state machine specifications for a small behavior routine. In this example the behavior of a robot soccer player is modeled which is supposed to search and approach a ball. To solve this task, the partial or primitive behaviors *seekBall*, *approachBall*, and *trackBall* are applied, which respectively let the robot search for the ball, move towards the ball, and track the ball with a camera located at the head of the robot. In both versions the robot will first assume that the position of the ball is unknown and search for it. When it finds it, the robot will concurrently move towards the ball and track the ball with the camera until the ball, and thus the target state of this behavior is reached.

Another comparable formal approach for behavior specification which is based on hybrid automata and supports model checking is described in [7]. A translator for creating specifications for the architecture presented in this paper from hybrid automata specifications has been developed [21].

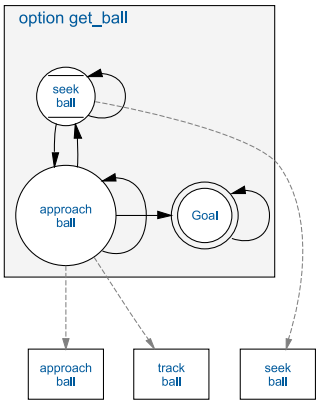


Figure 4: A possible adaptation of the Petri nets example using concurrent hierarchical state machines

3. THE EXTENSIBLE BEHAVIOR SPECIFICATION LANGUAGE (XABSL)

XABSL is a tool for implementing agent behavior using hierarchical state machines based on a pragmatic and formal approach. It consists of the following components:

- The modular behavior architecture based on concurrent hierarchical finite state machines,
- The specification language used for describing hierarchical state machines,
- A compiler generating documentations and intermediate code to be parsed by the runtime system,
- The C++ runtime library used to execute the behavior inside an agent software environment.

The first version of XABSL was developed by M. Lötzsch in 2002 [13, 14]. Since then it has been improved in several aspects, e.g. the original XML behavior description was replaced with the current specification language [15] and concurrent execution of state machines has been added [19].

3.1 Concurrent hierarchical finite state machines

The architecture applied in XABSL uses hierarchical finite state machines. The behavior of an agent is subdivided into simple state machines, known as *options*. These options are composed into a complex hierarchical state machine as described in the following. Each option defines a set of actions to be executed while each of its states is active. These actions can include other options, which are executed for as long as the respective state in the calling option is active. Through these calls from one option to another, the set of options is organized in a hierarchy. Similar to procedure calls, options can have parameters, where parameter values are specified by the calling option. Multiple options can also be called simultaneously resulting in the concurrent execution of these options. The current state of the hierarchical state machine is defined by the current states of the subset of currently active options, which form a tree, the *option activation tree*, starting with a distinguished *root option*. The

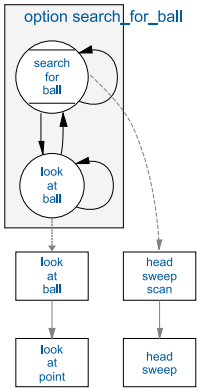


Figure 5: An example option from the robot soccer scenario. The circles represent the two states of this option. The circle with two horizontal lines is the initial state. The arrows between the states represent possible state transitions. The boxes below the option *search_for_ball* represent the subgraph of the option graph containing the options called subsequently. The dashed arrows show which options get executed in each state.

set of options forms an acyclic directed graph, which is called an *option graph*.

By employing a hierarchical structure, not only is it possible to compose complex behavior options from less complex subordinated options, but also reusing behavior options in different contexts is possible, since the same option can be referenced from different calling options. This allows the developer to modularize the behavior. For example commonly used and well-tuned behavior aspects can be placed in one option which then can be used in several different options without having to duplicate source code.

As above mentioned, the option graph is acyclic, i.e. recursive option calls are not allowed (directly or indirectly). Furthermore, it is illegal if the same option is called from different concurrently executed options, i.e. each option can only appear once in the current option activation tree.

The execution of the hierarchical state machines is performed using discrete execution cycles, which can occur after fixed time intervals, or can be coupled to events such as sensor cycle times. In each execution cycle the current state of each of the active options is determined, starting at the root option, subsequently checking each called option. Thus, the complete option activation tree is updated in every step. Whenever an option gets activated, which was not active in the previous execution cycle, e.g. when it is activated for the first time, the distinguished *initial state* of that option is activated.

The states of an option can be marked as *target states*. For instance, an option might reach a target state when the subtask the option is supposed to achieve has been carried out. A calling option can query whether a called option has reached a target state, and, in that case, might decide to change its state in order to execute another sub-option, which, for instance, carries out the next subtask.

An example of an option and its sub-options from a behavior for a controlling the camera direction in autonomous robot soccer is given in Fig. 5. Whenever the ball was detected in the camera image the head should follow the ball. This is done in the state *look_at_ball* which executes the option of the same name which in turn calls the option *look_at_point* in order to set the head joint values for letting the head of the robot look at the desired position. When the ball is not detected a scanning motion is started in the state *search_for_ball* by calling the option *head_sweep_scan*.

3.2 Interaction with the software environment

Since the XABSL state machine is responsible only for action selection of an agent, it is always embedded in the overall functional software environment. Responsibilities of the surrounding algorithms could include, e.g., processing sensor information, maintaining a world model, actuator control, and communications with other agents. Thus, an interface is required to provide input which the decision making can be based on and to deliver output such as action commands which describe the selected actions. The interface provided in the XABSL architecture makes uses of user-defined symbolic references to variables in the software environment and also enables to directly execute already implemented behavior routines from inside the hierarchical state machines.

3.3 Specification language

In order to have a compact and easy-to-understand method of describing XABSL behavior conveniently a description language was developed, replacing the XML dialect which was used before. Fig. 6 shows example source code. Using the XABSL compiler the source code can be compiled into an intermediate code, which can be parsed easily by the runtime library. Furthermore the compiler is able to generate symbol files for code completion and syntax highlighting in various editors and an XML representation can be generated. The XML representation can be processed further using standard techniques such as XSLT processors e.g. in order to generate documentation.

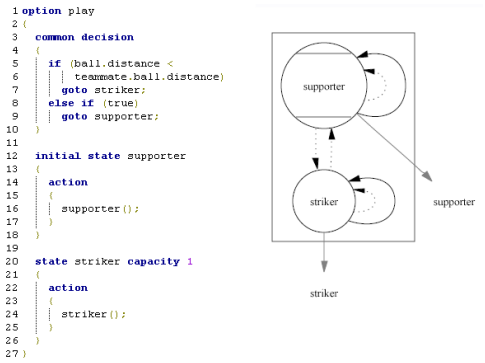
Expressions can be specified which are used in conditions in decision trees, parameter values, and output symbol assignments. They can combine references to option parameters, input symbols, output symbols (which evaluate to the last value set during behavior execution), enumerations using arithmetic and logical operators.

3.4 Cooperative Multi-Robot Systems

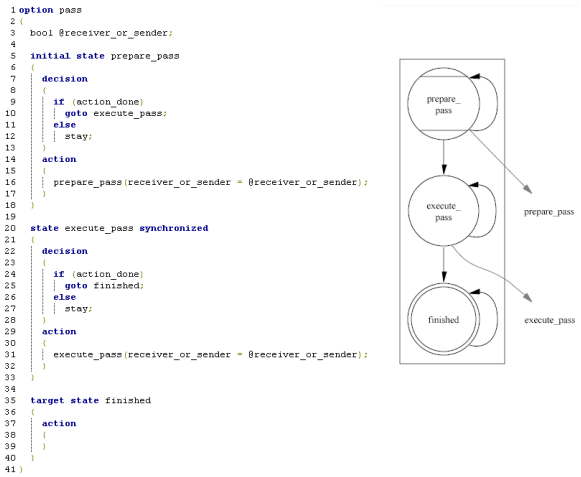
XABSL provides the following features to support cooperation between multiple communicating agents:

- A typical requirement is that a certain state of a state machine can only be executed by at most a given number of agents at the same time. The maximum number of agents that can execute a state is called the capacity of the state. A possible example might be a team of robots navigating through a narrow passage which can only be entered by a certain number of robots at once without blocking each other. Another example and its implementation in XABSL is shown in Fig. 7(a).

- Another requirement is that the actions of multiple agents might need to be synchronized. This can be realized by specifying, that all agents currently executing an option are required to enter a certain state of the option at the same time. If an agent tries to enter the state it will wait until every other agent is also ready to enter the state. A XABSL example is shown in Fig. 7(b). Optionally a minimum number of agents that are required to enter the synchronized



(a) State *striker* has capacity of one



(b) State *execute_pass* is executed synchronized

Figure 7: Example a) shows a state machine for role assignment in the robot soccer scenario. Only the field player which is closest to the ball shall attack the ball. Therefore the state *striker* has a capacity of one. Example b) also comes from the field of robot soccer: It shows an option for pass play. Only after both robots are finished preparing for the pass, e.g. aligning towards the other player, they will enter the state *execute_pass* synchronously.

state can be specified. The set of agents which execute an option synchronously can also be a subset of all the agents of a cooperative scenario. Only the agents that are currently executing the behavior option which requires synchronization are taken into consideration.

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

In a typical approach for realizing complex multi-robot applications using XABSL there is a single hierarchical behavior which is executed by all of the robots. The behavior can contain state machines for task allocation in the top levels of the hierarchy. In case of heterogeneous multi-robot applications, the different capabilities and limitations of each robot can be made available to the behavior through input symbols. According to these symbols the robot specific subbehaviors can be selected.

In most multi-agent environments, e.g. in every real multi-robot application, one cannot assume, that messages between agents will be sent and received instantaneously. Therefore, conflicts may arise, e.g. when two agents try to enter a state with a capacity of one nearly at the same time. In order to prevent such conflicts, some form of negotiation is necessary. In the proposed extension of XABSL the following negotiation pattern is applied: Whenever an agent tries to enter a state with a capacity it signals this to other agents and waits for a certain amount of time before entering the state. If the number of agents trying to enter exceeds the available capacity of a state, a user-defined agent prioritization is applied. It is easy to see, that increasing this delay leads to an increased protection against capacity conflicts. Only if the delay time is greater or equal to the maximum round trip time of sending a message to all other agents and receiving respective responses, it is guaranteed that the number of agents executing a state will never exceed the capacity of the state (cf. Fig. 8). On the other hand increasing the delay time leads to a reduced reactivity of the state machines. Thus, there is a trade-off between prevention of possible conflicts and reactivity. In some applications it might be critical to guarantee that the capacity of a state never gets exceeded, not even for very small amounts of time. In other applications it might be more important, that decisions are made as fast as possible (e.g. in the robot soccer scenario). Therefore the delay time is a parameter selectable by the application programmer.

In order to show that multi-robot applications can be realized easily using these features an example application has been implemented. The scenario of the application is the 2007 Passing Challenge of the RoboCup Four-Legged League. Three Sony Aibo robots are supposed to pass an orange ball back and forth between each other (cf. Fig. 10). In this example both presented features for the specification of cooperative behaviors are used. Utilizing a capacity state a task assignment is realized similar to the one given in the first example (cf. Fig. 7(a)) in order to decide which of the robots will go to the ball and catch it while the others wait until they receive a pass. When performing the pass both robots synchronize their actions as described in the previous example (cf. Fig. 7(b)). Fig. 9 shows the option graph used for this application. This implementation also is a good example for the support of code reuse as most of the required options could be taken from the standard robot soccer application such as behaviors for controlling the ball.

4. APPLICATIONS

Although XABSL was developed in the RoboCup robot soccer context as the behavior architecture successfully applied by the *GermanTeam* [20] in the Four-Legged Robot League since 2002 [13, 14]. It is not limited by any means to the robot soccer domain or related applications. It is

search_for_ball.xabsl

```
/** Scan mode for looking at the ball/searching for the ball */
option search_for_ball
{
  initial state search_for_ball
  {
    decision
    {
      if (ball.was_seen)
        goto look_at_ball;
      else
        stay;
    }
    action
    {
      head_sweep_scan(start_left_side = ball.angle > 0);
    }
  }

  state look_at_ball
  {
    decision
    {
      if (ball.time_since_last_seen > 500)
        goto search_for_ball;
      else
        stay;
    }
    action
    {
      look_at_ball();
    }
  }
}
```

Figure 6: Example XABSL source code of the option *search_for_ball* consisting of two state definitions. Each state definition contains two parts. The first is the specification of the decision tree, describing the transitions to other states and the conditions under which they are taken. The second part describes the actions to be performed while a state is active, consisting of option (or basic behavior) calls and output symbol assignments. The code is shown in the XABSL editor, which is available for download on the XABSL website [16].

not even limited to robot applications at all, but can rather be applied for describing the behavior of arbitrary software agents. RoboCup serves as a common testbed and benchmark for research in different fields of robotics and artificial intelligence.

In the Four-Legged Robot League teams of four of Sony's AIBO robots play soccer autonomously against each other. One of the major challenges in this league is the limited onboard computational power and the noisy and unreliable perception based on a directed camera with a low resolution of 208×160 pixel and an opening angle of 45 degrees, resulting in a very limited field of view, perceiving only small portions of the playing field at once. Another major challenge is four-legged locomotion, which generates a high amount of uncertainty in the actions of the robot. XABSL has proven successful in dealing with these kinds of uncertainties, as it has been applied for complex competitive soccer playing behaviors, which have helped the *GermanTeam* to become world champion in the Four-Legged League twice in 2004 and 2005.

The successful application of XABSL in the *GermanTeam* has also shown that the modular architecture supports the cooperative development of a large team of robot programmers on a complex project. Options can be developed and tested independently and new options are easily integrated in an existing behavior.

In RoboCup XABSL is not only being applied by a large number of teams in the Four-Legged Robot League, but also on different robots by teams in all robot leagues: In the Humanoid Robot League XABSL is applied by several teams including [6]. In the Middle Size Robot League the team *CoPS* from Stuttgart developed a graphical behavior modeling tool using Petri Nets which can automatically generate XABSL source code [23]. The Small Size Robot League team *B-Smart* uses XABSL to control the behavior of their robots [3].

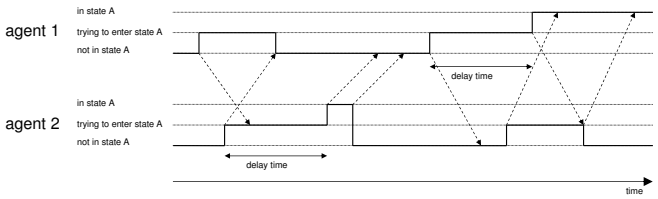


Figure 8: An example of the coordinated execution of a state with a capacity of one by two agents. Agent 2 has higher priority than agent 1. The dashed arrows depict the amount of time required for signaling a state change to the other agent. The delay time before entering the state is chosen as twice the amount of time required for transferring a message from one agent to another. Therefore, there are no capacity conflicts.

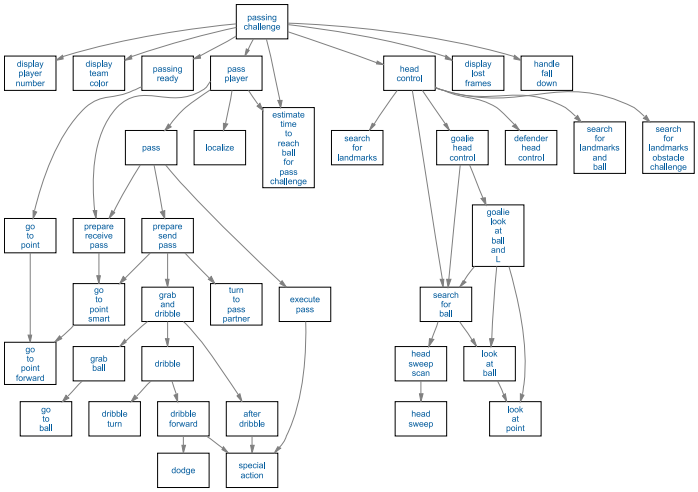


Figure 9: The option graph of the Passing Challenge example

Integration of learning approaches must be possible in a formal behavior specification method suited for complex multi-robot applications. Some subtasks in the robot soccer scenario are very well suited for applying machine learning approaches. In the Four-Legged Robot League ball grabbing behavior has successfully been improved through policy gradient learning [5]. Applying such approaches to subtasks of

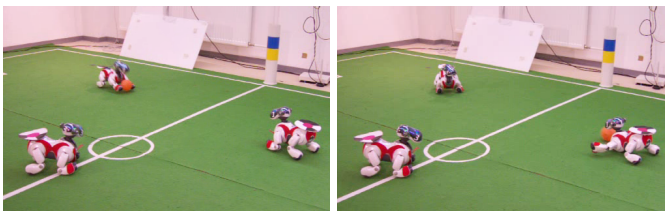


Figure 10: A successful pass between two four-legged robots (cf. Figs. 7, 9)

an agent behavior specified in XABSL is supported very well due to the hierarchical decomposition. In a hierarchical state machine different types of behavior programming, such as machine learning approaches, can be applied easily on different layers of the hierarchy. For instance, the optimal parameters of the ball grabbing behavior of the *GermanTeam* were optimized semi-automatically using Asynchronous Parallel Pattern Search [9, 11].

An application outside of the robot soccer domain has been realized successfully in a case study of cooperating, strongly heterogeneous, autonomous robots: the humanoid robot *Bruno* and a *Pioneer 2DX* wheeled robot [10].

A small sample application, which is an example agent for the ASCII Robot Soccer [2] simulation, is available online on the XABSL website [16].

5. CONCLUSIONS

In this paper a comparison of the extensible agent behavior specification language XABSL with Petri Net Plans and COLBERT has been made by means of basic examples for robot behavior. Furthermore, the extension of XABSL to multi-robot systems has been presented and illustrated by complex examples from robot soccer. Also, a large number of successful applications in different domains from different groups demonstrate that the approach is well suited for realizing complex real-world robot applications. The newly introduced features present a convenient and structured method of organizing multi-robot cooperations.

Ongoing research aims at investigating how behavior programming can benefit from the further use of formal methods, for example in order to prove certain properties of implemented behaviors. Also there will be continued investigation of application domains outside of robot soccer and large, complex multi-robot cooperations.

6. REFERENCES

[1] B. Bakker and J. Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of IAS-8*, pages 438–445, 2004.

[2] T. Balch. The ASCII robot soccer homepage, 1995. http://www-2.cs.cmu.edu/~trb/soccer/.

[3] O. Birbach, A. Burchardt, C. Elfers, T. Laue, F. Penquitt, T. Schindler, and K. Stoye. B-smart. team description for robocup 2006. In *RoboCup 2006: Robot Soccer World Cup X Preproceedings*, 2006.

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

[5] P. Fidelman and P. Stone. The chin pinch: A case study in skill learning on a legged robot. In *RoboCup-2006: Robot Soccer World Cup X*, pages 59–71. Springer Verlag, 2007.

[6] M. Friedmann, J. Kiener, S. Petters, H. Sakamoto, D. Thomas, and O. von Stryk. Versatile, high-quality motions and behavior control of humanoid soccer robots. In *Proc. Workshop on Humanoid Soccer Robots of the 2006 IEEE-RAS Int. Conf. on Humanoid Robots*, pages 9–16, Genoa, Italy, 2006.

[7] U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg. Hybrid multiagent systems with timed synchronization – specification and model checking. In *Proceedings of 5th International Workshop on Programming Multi-Agent Systems*, pages 170–185, Honolulu, Hawaii, 2007.

[8] 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.

[9] G. A. Gray and T. G. Kolda. Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software*, 32(3):485–507, 2006.

[10] J. Kiener and O. von Stryk. Cooperation of heterogeneous, autonomous robots: A case study of humanoid and wheeled robots. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 959–964, San Diego, USA, 2007.

[11] T. G. Kolda. Revisiting asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Optimization*, 16(2):563–586, 2005.

[12] 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.

[13] 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.

[14] 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.

[15] M. Lötzsch, M. Risler, and M. Jüngel. XABSL - A pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, 2006.

[16] M. Lötzsch, M. Risler, and M. Jüngel. XABSL web site, 2007. http://www.xabsl.de.

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

[18] D. McDermott. PDDL — the planning domain definition language. Technical report, Yale Univ., 1998.

[19] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. GermanTeam 2007 - The German national RoboCup team. In *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[20] T. Röfer et al. GermanTeam RoboCup 2005. Technical report, HU Berlin, Universität Bremen, TU Darmstadt, Universität Dortmund, 2005. Available online: http://www.germanteam.org/GT2005.pdf.

[21] F. Ruh. A translator for cooperative strategies of mobile agents for four-legged robots. Master thesis. Hochschule Harz, 2007.

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

[23] O. Zweigle, R. Lafrenz, T. Buchheim, U.-P. Käppeler, H. Rajaie, F. Schreiber, and P.Levi. Cooperative agent behavior based on special interaction nets. In *IAS*, pages 651–659. IOS Press, 2006.