Translating Cooperative Strategies for Robot Behavior*

Florian Ruh and Frieder Stolzenburg

Hochschule Harz, Automation and Computer Sciences Department, D-38855 Wernigerode
{fruh,fstolzenburg}@hs-harz.de

Abstract. This paper presents a method for engineering and programming multi-robot systems, based on a combination of statecharts and hybrid automata, which are well-known in the fields of software engineering and artificial intelligence. This formal specification method allows graphical presentation of the whole multiagent system behavior. In addition, these specifications can be directly executed on mobile robots. We describe the transformation process from the specification to executable code, after introducing the necessary definitions. A translator that automatically converts hybrid hierarchical statecharts into simple flat hybrid automata (i.e. without hierarchies) has been implemented. The respective tool allows the text-based input of hybrid hierarchical automata specifications of multiagent system with synchronization. The translation into flat automata is performed by means of different plug-ins, leading e.g. to executable code for Sony Aibo robot dogs. The plug-in just mentioned has been successfully applied in the RoboCup four-legged league.
Key words: agent-oriented software engineering; multiagent systems; RoboCup; tools for intelligent systems.

1 Introduction

Robotic soccer provides many research challenges and one of them is behavior control including the subjects of team play, cooperation and flexible, quick reaction. A soccer team can be designed as a homogeneous multiagent system. Since the behavior of multiagent systems and agents alone can be understood as driven by external events and internal states, an efficient way to model such systems are state transition diagrams, which are well-established in software engineering. They are graphical representations of finite state machines with hierarchically structured states and transitions which lead from one state to another depending on the input or events. Outputs or actions can be done during transitions or in states. State transition diagrams have been applied successfully for multiagent systems in general and in the RoboCup, a simulation of (human) soccer with real or simulated robots (see e.g. [2, 5, 19]).

However, state transition diagrams do not properly cover all aspects of multiagent systems. Therefore, hybrid hierarchical automata (HHA) with timed synchronization have been developed to take continuous processes in the environment into account [7]. Moreover, they can consider time as an additional factor for synchronization processes. This formalism can help to model situations when two or more agents have to deal with one resource. In the domain of soccer e.g., the agents have to consent that exactly one player goes to the ball.

* This paper emerged from the master thesis of the first author [16].

Therefore, as a running example, we consider a scenario influenced by the UEFA champions league competition 2006/2007: the *Makaay move* (see Fig. 1). Let there be one player of type *A* and two players of type *B* in the offensive team. Player *A* performs the kick-off, while the players of type *B* are waiting in different sectors on the pitch, which is divided into sectors (cf. [6]). Player *A* chooses a direction for passing (right or left midfield, sectors 3 or 5), then kicks off and passes to one player of type *B* in the destination sector. Player *A* runs to sector 1 (middle offense) where *B* has passed the ball to. Finally, player *A* tries to shoot to the goal directly. If it fails, *A* tries it again. Meanwhile, *B* goes to the ball if it is nearer to it. *B* then passes to *A* in sector 1 again.

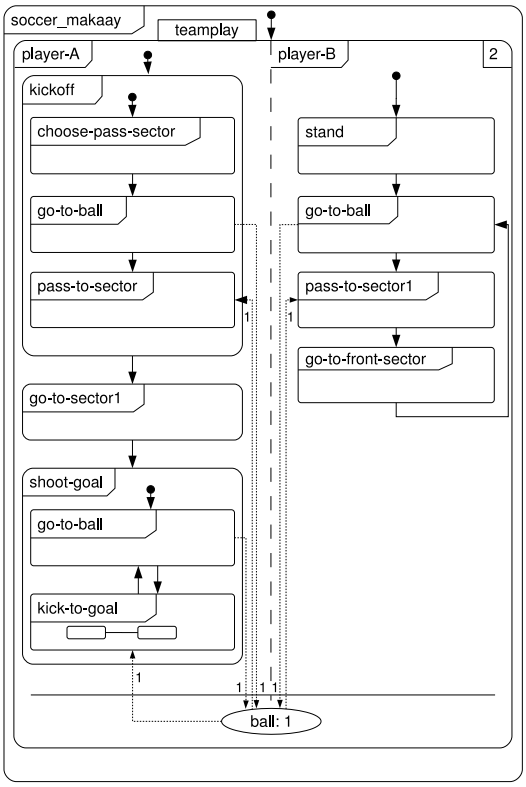


Fig. 1. Statechart for Makaay move.

In the sequel, Sect. 2 covers the formal specification of hybrid statecharts. Corresponding description and target languages are defined and compared in Sect. 3. With these foundations, we can create a concept for the translation process. Sect. 4 then deals with the design of the application and shows an example of use. Finally, we discuss related works in Sect. 5 and conclude with Sect. 6.

2 Hybrid Statecharts

2.1 States and Transitions

In a realistic physical environment, it is inevitable to consider continuous actions in addition to discrete changes. Hybrid automata extend regular state transition diagrams with methods that deal with those continuous actions. To understand the characteristics, we will introduce several definitions for hierarchical hybrid automata with timed synchronization now [7, 8] – called HHA.

Definition 1 (basic components). *The basic components of a* state machine *are the following disjoint sets:*

S**:** *a finite set of* states*, partitioned into three disjoint sets:* S_{simple}*,* S_{comp}*, and* S_{conc} *— called simple, composite and concurrent states, containing one designated* start state $s_0 \in S_{comp} \cup S_{conc}$*;*

X**:** *a finite set of variables, partitioned into two disjoint sets:* X_{real} *and* X_{int} *— the continuous/real-numbered and the integral/integer variables, respectively; for each* $x \in X$ *we introduce the variables* x' *for the conclusions of a discrete change;*

T: *a finite set of transitions with* $T \subseteq S \times S$*.*

Definition 2 (state hierarchy). *Each state s is associated with zero, one or more* initial states $\alpha(s)$*: a simple state has zero, a composite state exactly one, and a concurrent state more than one initial state. In the latter case, the initial states are called* regions*. Moreover, each state* $s \in S \setminus \{s_0\}$ *is associated to exactly one superior state* $\beta(s)$*. Therefore, it must hold* $\beta(s) \in S_{conc} \cup S_{comp}$*. A concurrent state must not directly contain other concurrent ones. Furthermore, it is assumed that all transitions* $s_1 T s_2 \in T$ *keep to the hierarchy, i. e.* $\beta(s_1) = \beta(s_2)$*. Furthermore, we write* $\alpha^n(s)$ *or* $\beta^n(s)$ *for the n-fold application of* α *or* β *to s, in particular,* $\alpha^0(s) = \beta^0(s) = s$*. Variables* $x \in X$ *may be declared locally in a certain state* $\gamma(x) \in S$*. A variable* $x \in X$ *is* valid *in all states* $s \in S$ *with* $\beta^n(s) = \gamma(x)$ *for some* $n \geq 0$*, unless another variable with the same name overwrites it locally.*

As said earlier, Fig. 1 depicts the statechart for our running soccer example. Here, states are named after their affiliation to the players or the actions which are being done at that moment. The states *soccer_makaay* (which is the start state s_0 here), *kickoff*, *go-to-ball*, *player-A* and *player-B* are composite states; *teamplay* is a concurrent state while all others are simple states. The oval symbol *ball* is a synchronization point and will be discussed in Sect. 2.2.

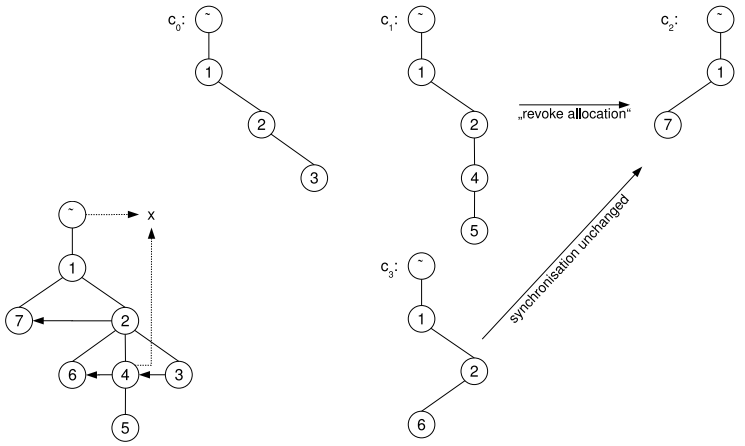
Definition 3 (jump and state conditions). *For each transition, there exists a* jump condition*. This is a predicate with free variables from the valid variables of* $X \cup X'$*. Additionally, each state* $s \in S$ *contains a state condition which describes continuous changes in s. It is a predicate with free variables from* $X \cup \{t\}$*.*

Events are well-known in UML statecharts [12] and hybrid automata [8]. They can easily be expressed by (binary) integer variables in our formalism. Therefore, we do not introduce them explicitly in our definitions. But in contrast to simple hybrid automata, we introduce hierarchies. Fig. 2(a) shows an example state tree, which is induced by the β-function. Here, R is the root of the tree, and e.g. state 1 can be reached from 5, i.e. $1 = \beta^3(5)$. Note that the value of β^n is always uniquely determined due to the tree-like (and not graph-like) structure of the state hierarchy. Furthermore, let $\alpha^3(R) = 3$. A configuration (defined next) is the subset of the active states in the state tree.

Definition 4 (configuration and completion). *A* configuration *c is a rooted tree of states with the root node as the topmost initial state of the overall state machine. Whenever a state s is an immediate predecessor of* s' *in c, it must hold* $\beta(s') = s$*. A configuration must be* completed *by applying the following procedure recursively as long as possible to leaf nodes: if there is a leaf node in c labeled with a state s, then introduce all* $\alpha(s)$ *as immediate successors of s.*

The semantics of our automata can now be defined by alternating sequences of discrete and continuous steps. Following the synchrony hypothesis, we assume that discrete state changes (via transitions whose annotated jump condition holds in the current situation) happen in zero time, while continuous steps (within one state) may last some time. Due to the lack of space, for details on the semantics of HHA, the reader is referred to [15].

Fig. 2(b) demonstrates the relationship between state trees and configurations. It depicts several configurations that are created from the state tree in Fig. 2(a). A configuration itself can be connected to another one. The original transition t, which was used for the completion of s_2 in c_2, is used in a discrete step while its origin is changed from s_1 to c_1 and its target from s_2 to c_2.



(a) State tree with synchronization point x.

(b) Configurations with synchronization problem.

Fig. 2. State tree and configurations of an automaton.

2.2 Synchronization

Synchronization is significant for modeling multiagent systems. Usually, a system deals with limited resources. The interaction with them can take part in several states. Especially when reacting to events from the environment, the reaction process takes some time $\tau > 0$. For this, a *synchronization* takes care of the common resources defined at a *synchronization point*. While synchronization is associated with transitions, implemented via labels in original hybrid automata [8], synchronization is associated with states in HHA, i.e. actions which last some certain time. In contrast to this, the synchrony hypothesis states (for discrete steps), that a system is infinitely fast and therefore can react immediately within zero seconds, i.e., a transition takes zero time.

Definition 5. *A synchronization point is identified by a variable $x \in X_{sync} \subseteq X$ with a maximum capacity $C(x) > 0$. Each state connected to the synchronization point is*

classified by one of the following relations, $R_+ \subseteq S \times X_{sync}$ or $R_- \subseteq X_{sync} \times S$. If a state increases the capacity, it will be classified by R_+ and otherwise by R_-, if it decreases it (or resets the resource). In general, each connection in $R_+ \cup R_-$ is annotated with a number m with $0 < m \leq C(x)$ which identifies the volume to be increased or decreased from the synchronization point, respectively.

Synchronization may take some time, since they are connected to (continuous) states and not to discrete transitions. Thus, the synchronization process can theoretically be interfered by other actions or concurrent states which also try to share the same synchronization point. To avoid side effects that may lead to inconsistency or even system failure, the process is separated into allocation and (future) occupation of resources. For this, the allocation variables x_+ and x_- register the request for occupation or release for each synchronization point x. Therefore, x_+ and x_- must be added to X.

In this case (synchronization point x and connected state s), $s_1 T s_2$ is called *incoming transition* for s iff $\alpha^n(s_2) = s$ for some $n \geq 0$, *initializing transition* iff it is an incoming one with $\alpha^n(s) = \gamma(x)$, *outgoing transition* iff $s_1 = \beta^n(s)$ for some $n \geq 0$ where s_1 occurs in the current configuration and x is valid in s, *successful outgoing transition* iff it is an outgoing transition with $s_1 = s$ and *failed outgoing transition* iff it is not a successful outgoing transition. Note that outgoing transitions cannot be characterized statically but only dynamically by investigating the configuration trees. This is an important issue for the revoking of the allocation (see Sect. 4.2). At a synchronization point x, additional constraints must be defined which affect the transitions that are incident with all states s connected to x. Due to the lack of space, for details on the synchronization concept, the reader is referred to [7].

The synchronization point *ball* in Fig. 1 has a capacity of 1. Thus, it can be interpreted as a Boolean value as there is only one ball in a soccer match. Both states *go-to-ball* occupy the synchronization point *ball*. Hence, they belong to the relation R_+. The states *kick-to-goal* and *pass-to-sector* release it and therefore belong to R_-.

The example in Fig. 2(a) also makes use of a synchronization point. As seen in the tree, the state R introduces the synchronization point x while 4 is somehow using it here. The definition is marked with the dashed arrow pointing at x. However, for some multiagent systems, the synchronization must be converted into ordinary variables if a target platform does not provide synchronization interfaces.

3 Specification Languages

After having defined basic concepts, let us now consider concrete languages for programming multiagent systems with HHA. Therefore, we will discuss two languages briefly in the sequel: HAL and XABSL.

The project goals of HAL [3] were the definition of an ASCII-formatted specification language for hybrid automata with timed-synchronization and, furthermore, its transformation into an input format for model checkers such as *HyTech* [8]. HAL is at the same time the name of the project and the name for the specification language (*Hybrid Automaton Language*). This corresponds to the definitions introduced in the previous section. A HAL specification is usually written into an ASCII formatted file.

According to the syntax, it consists of a global frame which must be a composite automaton. It may include several other automata following the rules of hybrid automata with timed synchronization. Even though the terms of inheritance, polymorphism are not defined in HAL syntax, modularization is actually known. The namespace of two parallel automata cannot collide while subsequent automata can access variables of their superiors. An example is shown in the listing (Fig. 3).

```
composite makaay {
  start( teamplay );
  concurrent teamplay {
    syncpoint( ball, 1 );
    region player_A {
      cardinality := 1;
      start( kickoff );
      composite kickoff {
        start( choose_pass_sector );
        var pass_sector := 0;
        var random_05_3_5 = 0;
        simple choose_pass_sector {
          flow       := pass_sector~ = random_05_3_5;
          invariant := pass_sector != 3 & pass_sector != 5;
          trans      := ( go_to_ball, pass_sector == 3 |
            pass_sector == 5 );
        } % choose_pass_sector
        simple go_to_ball {
          sync( ball, 1 );
          flow := go_to_ball_without_turning_maxspeed120;
          invariant := ball_seen_distance >= 70;
          trans      := ( pass_to_sector, ball_seen_distance < 70
             );
        } % go_to_ball
        % (...)
        invariant := ball_sector == 4;
        trans      := ( go_to_sector1, ball_sector != 4 );
      } % kickoff
      % (..)
    } % player_A
    % (...)
  } % teamplay
} % makaay
```

Fig. 3. HAL specification.

Another successful approach of modeling agent behavior is *XABSL* (Extensible Agent Behavior Specification Language) [11]. It was developed and integrated into the code basis of the *GermanTeam*, several times world and German champion in the

RoboCup four-legged league, as a language for behavior engineering. The specifications can be transformed automatically into intermediate code which has to be interpreted on the target platform by the *XabslEngine*. The XABSL package also provides functionalities for visualization, debugging and documentation. The *option* division in XABSL specifications includes a global symbol file to get access to the environment. It consists of one initial and several other states with their own decision trees. The *action* division specifies all assignments that are executed there. A subsequent option call is also possible.

4 The Translator Tool

The HAL converter provides a window-based flattening mechanism for state machine specifications, a batch mode for quick processing, and re-usability. Additionally, there should be a graphical editor to easily create source code from hybrid statecharts. Already created files (or files that are created manually) are allowed to be used as an input for the application. Hence, a lexer and a parser provide the conformity with the HAL syntax. With this design, it is possible to create a hybrid automaton which can be used later as input for the flattening algorithm (see below). The translator from HAL to XABSL shall cover all features of synchronized hybrid state machines that can be transferred to XABSL.

4.1 Flattening Algorithm

For the translation process, there is no simple one-to-one structural mapping between HHA and XABSL. As XABSL and also standard verification tools often are not able to cope with hierarchies, it is required to flatten the automaton, i.e., all states except the initial one are transformed into simple ones. As the translator shall be feasible of creating processable output for those tools, this gives us another reason to flatten the hierarchical structure. Though this transformation may lead to state explosion, it could be avoided, nevertheless, if hierarchical configurations could be processed as directly as in some logic-based implementations [7, 15].

In the implementation, configurations are used to clarify which agent currently is in which state. The flattening algorithm processes an input state tree and converts it to a set of configurations. In particular, the output can be used to simplify the agent's behavior structure and to gain performance due to less complexity. For this, the algorithm is divided into four major parts.

1. **Copy regions**
   Expand the regions in the tree according to their cardinality c (given in the upper right corner of a region). Modify each region to a composite state, copy it c-times and replace the original with the copies.
2. **Globalize variables and constants**
   Each state may introduce variables and constants. Each local definition must be globalized as it will be used in the configuration flows and transitions later on. The global definitions must be uniquely named to avoid namespace collisions.

3. **Convert synchronizations**
   If a state uses a synchronization point to interact with other states, these synchronizations must be resolved. Due to their complexity, a relatively extensive inspection is required which is explained in detail in Sect. 2.2 and 4.2. Although the resulting additions to transition guards reduce readability, the even more complex process of inter-state synchronization could be eliminated. A practical approach for the detection of the correct place to revoke an allocation is given below.
4. **Create configurations**
   Each state tree possesses an initial configuration c_0. This contains all the initial states that can be reached in the tree beginning at the root. According to the completion algorithm (Def. 4), the configurations are created recursively beginning at c_0. Already existing configurations will be recognized and used if transitions lead to them. These newly created transitions form the discrete steps of the system.

The synchronization conversion in the third step is a rather complex process. At first, all synchronization points in the automaton are collected. After this, the automaton will be traversed, and the occupation, the release, as well as the allocation, and its revoke are added for each synchronization found in the state tree. The synchronization point x itself, its maximum capacity $C(x)$, and its allocation variables x_+ and x_- are converted into global variables. For each transition type, different expressions must be added to the guards and the discrete expressions. However, a flag x_f for each synchronization point x is introduced indicating its current status. If x is occupied then $x_f := -1$. If x is allocated but not occupied yet then $x_f := 1$. Otherwise, $x_f := 0$. For all not initializing incoming transitions, $x_f := 1$ will be added to their discrete expressions, $x_f := 0$ will be added for all initializing incoming transitions, $x_f := -1$ will be added for all successful outgoing transitions. For each not successful outgoing transition, it must be checked if x is already allocated but not occupied by this synchronization. Therefore, the transition must be duplicated. The comparison $x_f = 1$ is added to the guard of the first transition, $x_f \neq 1$ is added to the second one. The revocation of the allocation is added only to the discrete expression of the first one. Finally, all synchronization points can be erased as they are now properly converted into ordinary variables.

4.2 Allocation in Synchronizations

During the development of the theoretical model of hybrid automata with timed synchronization, a problem concerning not successfully outgoing transitions occurs. The correct situation has to be found, when the allocation shall be revoked, since it must actually be done only once per occupation. For this purpose, some definitions have to be introduced.

Let $\delta(s)$ return all variables used in the state s but not defined there. Furthermore, we introduce a mapping ζ which returns all state successors of s that use x and are part of the configuration c:

$$\zeta(s,x,c) = \{s_i \mid \beta^n(s_i) = s \wedge x \in \delta(s_i) \wedge s_i \in S(c), n > 0\}$$

Fig. 2(a) depicts a simple example for that synchronization problem. The dashed arrows indicate the definition and the usage of the synchronization point x. The state tree

shows – among others – a transition from state 2 to 7. Fig. 2(b) shows the appropriate configurations with c_0 being the initial one. In this case, the synchronization point x is defined in the state R while only 4 is using x. In fact, R must be a concurrent state as it defines a synchronization point. Though concurrent states usually have two or more regions, this example reduces complexity and actually uses only one.

Let us now have a closer look on what is happening in c_1 when the process has activated state 5. State 4 is also active as it is the immediate predecessor of state 5 in the tree. The transition from 2 to 7 is a not successful outgoing transition for 4 as $2 = \beta^n(4)$ with $n = 1 > 0$.

Now, to collect all states that may have allocated x before the transition t induces a discrete step to c_2, the mapping ζ can be applied. Here, ζ is used with the parameter 2 as this state is the origin of the transition. In the configuration c_1 this is a set containing one single state: $\zeta(2,x,c_1) = \{4\}$. That statement confirms that (only) 4 has allocated the synchronization point x in this situation. There has no occupation been done yet. So for the transition t, the allocation has to be revoked and further actions can be done during the process. On the contrary, the configuration c_3 does not have an active allocation or occupation since $\zeta(2,x,c_3) = \emptyset$. Therefore, the synchronization point remains unchanged for the transition t.

4.3 User Interface

From a shell the user can start the Java application in console or window mode with several mandatory and optional parameters. As shown in Fig. 4, all configurations can be set intuitively in the window mode. The required input file can either be typed into the textfield on the top of the main content pane or it can be chosen by using a file dialog window. The temporary and the target output file are named accordingly. However, they can be defined individually, too.

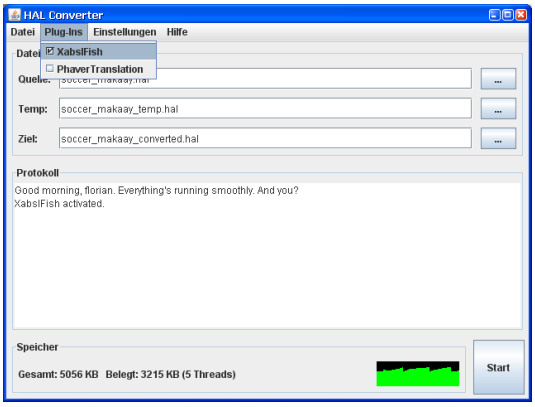


Fig. 4. HAL screen-shot before starting the translation.

4.4 XabslFish

The XabslFish plug-in defines constraints which check the input automaton for the proper structure. For this, the flattener algorithm must have processed the automaton. The regions must be copied according to their capacities, the variables must be globalized to provide an efficient handling of the XABSL symbol file and the synchronizations must be converted into ordinary variables. The variables must not be renamed during their globalization as they will later be translated by use of the configuration file.

The conversion of the flattened automaton starts with reading the appropriate configuration file. This is used to transform HAL variables to expressions, to basic behavior

calls or for their declaration in the output symbol file. In the second step of the translation, the symbol file is generated. A Boolean flag indicates if any symbol has to be written at all. In case that there is no symbol to write, the file will not be created. In consequence of that it can be decided if the future options will include this file or not. The third step is the major part of the translation. Here, the automaton tree is traversed and each node will be converted into an option. The subsequent automata become accessible via internal states while initial subsequent states keep their status. Each transition from a successor to another automaton is implemented into the state decision tree where discrete expressions are converted into actions in the target state. If all successor nodes are completed the own flow expressions of the current automaton will be converted into actions. This algorithm is processed recursively for each automaton.

4.5 An Example of Use

Let us now come back to our running example (Fig. 1). For purpose of clarity, the transition labels with the jump conditions and the discrete expressions are omitted as well as the flow expressions and invariants in the states. However, within this example there are several main features of synchronized hybrid automata covered. The soccer team has at least three players: one of type *A*, two of type *B*. Furthermore, there is exactly one ball on the field which can be interpreted as a resource with the maximum capacity 1. Due to the lack of space, for further details on the implementation, the reader is referred to [16].

5 Related Work

There are many related works on the specification of multiagent systems and also on software engineering of multiagent systems (see e.g. [2]), including Agent UML [13], where UML statecharts for modeling agent behavior are also considered, but not in the main focus of interest, however. We will therefore only briefly discuss some work on multi-robot coordination architecture, coordination mechanisms, and on formal specification of multi-robot systems.

As proposed by [17], multiagent systems have to deal with allocation and synchronization of tasks and concurrent subtasks. There are market-based approaches which support the coordination of the robot teams while each robot is paid revenue for each accomplished subtask and otherwise incurs cost for allocating team resources. ALLIANCE [14] is the name of an architecture for fault tolerant multi-robot cooperation. With this, it is possible to create multiagent systems that can deal with failures and uncertainties in the selection and execution of actions and dynamically changing environment.

In [10], coordination mechanisms are introduced in a concrete multi-robot architecture. The scenario description language Q in [9] concentrates on social agents that interact with humans. However, these articles deal with teamwork behaviors and interaction rather than translating a formalism to a specific hardware platform. Nevertheless, modeling and implementing multiagent systems is proposed in [2, 5]. Though this is

based on UML statecharts, yet, we use hybrid automata with timed synchronization in addition, in order to construct those systems.

The paper [1] presents a case study in multi-robot coordination, employing linear hybrid automata. By a rectangular approximation of the physical environment, the geometric regions in which a robot reaches a given goal faster with the help of communication, can be computed. [19] employs Petri nets for the specification of multiagent plans. Here, synchronization also can be expressed quite naturally within the Petri net framework. The MABLE language [18] is based on BDI agents, described textually, and a tool for modeling and verifying multiagent systems. However, the focus in the papers just mentioned is on verification or analysis and not on implementation and generating executable code on mobile robots, whereas we apply standard software engineering methods to real robots in this paper.

6 Discussion and Conclusions

XabslFish allows the behavior control of Aibo robots (or any other XABSL-driven robots) to be designed as a multiagent system with formal methods. Therefore, the performance of the robot is enhanced. The safety of a correct behavior control is based upon the fact that the translation from HAL to XABSL can precisely be adjusted manually for each input automaton. The application *XabslFish* supports the modeling of hybrid automata with timed synchronization and translates the specification to an understandable format for the target platform. The soccer domain is used as an example for multiagent systems, which have to act autonomously in a dynamic environment.

XabslFish translates multiagent system specifications from the hybrid automaton language *HAL* to *XABSL*. It is designed as a plug-in for the application, the *HAL* converter, that deals with hybrid automata. Even though it can translate the major part of a state machine automatically, some individual mappings must be defined in a configuration file. In summary, this paper exemplifies that multiagent systems can be specified by formal methods based on standard modeling procedures (namely state machines). It is also demonstrated, how by transformation techniques executable code can be generated, running on a mobile robot (namely the Aibo robot).

Future work will concentrate on an implementation of the formalism with constraint logic programming (CLP), which can be used for both, engineering and analysis of multiagent systems, following the lines of [15]. This will lead to an even more realistic knowledge engineering system.

References

1. R. Alur, J. M. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems: A case study in multi-robot coordination. In *World Congress on Formal Methods (1)*, pages 212–232, 1999.
2. T. Arai and F. Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In Castelfranchi and Johnson [4], pages 11–18. Volume 1.
3. T. Bernstein, D. Borns, C. Colmsee, K. Czarnotta, H. Germer, N. Nause, M. Pacha, R. Thomas, A. Vellguth, T. Wiebke, and M. Windler. HAL – hybrid automaton language.

Technical report, Department of Automation and Computer Sciences, Hochschule Harz, 2006. Team project description (in German).
4. C. Castelfranchi and W. L. Johnson, editors. *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems*, Bologna, Italy, 2002. ACM Press.
5. V. T. da Silva, R. Choren, and C. J. P. de Lucena. A UML based approach for modeling and implementing multi-agent systems. *Autonomous Agents and Multiagent Systems*, 2:914–921, 2004.
6. F. Dylla, A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, S. Schiffer, F. Stolzenburg, U. Visser, and T. Wagner. Approaching a formal soccer theory from behaviour specifications in robotic soccer. In P. Dabnichcki and A. Baca, editors, *Computers in Sport*, pages 161–185. WIT Press, Southampton, Boston, 2008.
7. U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg. Hybrid multiagent systems with timed synchronization – specification and model checking. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-Proceedings of 5th International Workshop on Programming Multi-Agent Systems at 6th International Joint Conference on Autonomous Agents & Multi-Agent Systems*, LNAI 4908, pages 205–220, Honolulu, 2008. Springer, Berlin, Heidelberg, New York.
8. T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, NJ, 1996. IEEE Computer Society Press.
9. T. Ishida and S. Yamane. Introduction to scenario description language q. In *ICKS '07: Proceedings of the Second International Conference on Informatics Research for Development of Knowledge Society Infrastructure*, pages 137–144, Washington, DC, USA, 2007. IEEE Computer Society.
10. G. A. Kaminka and I. Frenkel. Integration of coordination mechanisms in the BITE multi-robot architecture. In *ICRA-07*, 2007.
11. M. Lötzsch, M. Jüngel, M. Risler, and T. Krause. XABSL: The Extensible Agent Behavior Specification Language. URI: http://www2.informatik.hu-berlin.de/ki/XABSL/, 2006.
12. Object Management Group, Inc. *UML Version 2.1.2 (Infrastructure and Superstructure)*, November 2007.
13. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at 17th National Conference on Artificial Intelligence*, pages 3–17, 2000.
14. L. E. Parker. Alliance: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automaton*, 1998.
15. C. Reinl, F. Ruh, F. Stolzenburg, and O. von Stryk. Multi-robot systems optimization and analysis using MILP and CLP. In P. U. Lima, N. Vlassis, M. Spaan, and F. S. Melo, editors, *Workshop 1: Formal Models and Methods for Multi-Robot Systems at 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 11–16, Estoril, Portugal, 2008. International Foundation for Autonomous Agents and Multi-Agent Systems.
16. F. Ruh. A translator for cooperative strategies of mobile agents for four-legged robots. Master thesis, Fachbereich Automatisierung und Informatik, Hochschule Harz, 2007.
17. A. T. Stentz, M. B. Dias, R. M. Zlot, and N. Kalra. Market-based approaches for coordination of multi-robot teams at different granularities of interaction. In *Proceedings of the ANS 10th International Conference on Robotics and Remote Systems for Hazardous Environments*, March 2004.
18. M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In Castelfranchi and Johnson [4], pages 952–959. Volume 2.
19. V. A. Ziparo and L. Iocchi. Petri net plans. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06*, pages 267–289, 2006.