From the Specification of Multiagent Systems by Statecharts to Their Formal Analysis by Model Checking: Towards Safety-Critical Applications

Frieder Stolzenburg[1] and Toshiaki Arai[2]

[1] Hochschule Harz (University of Applied Studies and Research), Automation and Computer Sciences Department, Friedrichstr. 57–59, 38855 Wernigerode, GERMANY, fstolzenburg@hs-harz.de

[2] Information Technology Dept., Mitsubishi Nuclear Fuel Co.,Ltd, 622-1, Funaishikawa, Tokai-mura, Ibaraki 319-1197, JAPAN, ara@mmc.co.jp

Abstract. In order to design and implement multiagent systems, the specification method should be as expressive and comprehensive as possible. Statecharts, which are capable of describing dynamic systems and are widely accepted in the computer science community, are applied and investigated for this objective.
In this paper, multiagent systems are studied in the domain of robotic soccer, where the behavior of agents including collaboration is specified by means of UML statecharts [7,8]. This method is also applicable to industrial software applications. For example, a network application can be designed and specified by means of the same method (see [1]). The approach enables not only standardized design of multiagent systems, but also almost automatic translation of the specification into a running implementation.
As a natural extension of this methodology for designing multiagent systems, formal system analysis and verification should be possible, in order to investigate important system properties. In this paper, we will show how specifications of multiagent systems based on UML statecharts can be verified, by employing model checking techniques. Hence, the proposed specification technique can be used for both, automated multiagent system implementation and analysis.

Introduction

Multiagent systems are expected to be the foundation of robust, interconnected and highly advanced computing methods. This new technology is currently being applied to various fields such as electronic commerce, robotics, computer mediated collaboration, ubiquitous computing, and others.

In previous work, we investigated two domains of applications and discussed how the behaviors of multiagent systems are designed and specified by means of UML statecharts [1]. The advantage of this methodology is that both, researchers and engineers can design and develop software systems more rapidly, communication among team members becomes easier and more reliable, and the

M. Schillo et al. (Eds.): MATES 2003, LNAI 2831, pp. 131–143, 2003.

integration of relatively large software systems is equipped with features of multiagent systems, by means of a standardized specification technique. In addition, our formalism based on statecharts allows us the verification and formal analysis of multiagent systems [13].

We think that in the development of multiagent systems, standardized software design, which also makes software verification possible, is an important matter, such that more secure and reliable systems can be designed and implemented. In this paper, we introduce application domains in Sect. 1, and study how the behavior of agents can be specified by statecharts and analyzed by model checking technology.

We state our method with UML statecharts for the specification of multiagent systems in Sect. 2. There, we demonstrate how the behavior of state machines can be described by configurations and steps. After that, we show some examples of the specification of multiagent systems in Sect. 3.

Model checking is a very useful technique to verify behavioral properties of finite state systems with concurrency. In Sect. 4, we show how multiagent systems, specified by UML statecharts, can be formally analyzed by using model checking techniques. The rigorously formal specification of multiagent systems allows us to translate them into logical structures (Kripke structures), where verification of system properties by means of temporal logics, e.g. CTL, is possible. We end up with concluding remarks and remarks on related works in Sect. 5.

1 Example Applications

In this section, we introduce two applications, namely robotic soccer and a synchronous work flow manager in some detail. Although both applications are different at first glance, they can be described in a similar manner, using the same method for specification. We will describe these domains as multiagent systems in this section and continue in Sect. 3, where we also address the problem of specifying the interaction of several, possibly heterogeneous agents.

1.1 Robotic Soccer

In the RoboCup initiative, the soccer game is chosen as a central topic of research, aiming at innovations to be applied for socially significant problems and in industry. In order to perform actually a soccer game for a robot team, various technologies must be incorporated including design principles of autonomous agents. The RoboCup consists of several leagues with real robots in different sizes or virtual, i.e. simulated robots.

The RoboCup simulation league offers a software platform for research on the software aspects of RoboCup. And in our context, the software design aspect is the most important one. In the soccer domain, usually all agents have an identical internal structure except for the goal-keeper. Therefore, it is quite natural to state the behavior of the agents within one state machine.

1.2 Work Flow with Concurrent Actions

Concurrent actions often arise in industrial applications, that are connected to each other. In general, it is difficult to specify dynamically changing behaviors. In order to design, develop and test such applications, often complicated procedures are needed. But concurrently acting entities can be regarded as agents. And the interesting thing is that, also in this case, system specification is possible by means of the same method as for robotic soccer.

Let us introduce now the meeting organizer application—a case study chosen from typical industrial applications—as a testbed for the specification of multiagent systems with dynamic behavior and interrelationships. The *meeting organizer agent* and the *schedule manager agent* are deployed in this scenario. The meeting organizer agent provides the following functions: arranging appointments and notify meetings. On the other hand, the schedule manager agent holds all *user agents'* schedules.

Without explicit intervention of users, the meeting organizer agent shall be capable of arranging a meeting appointment, provided that each user's schedule manager agent keeps appropriate schedule data, i.e., the schedule manager agent gathers necessary data prior to the arrangement of the meeting organizer. It is assumed that many users can exploit the system via a local area network without direct connection. A user who initiates the arrangement is called organizer, by whom the desired meeting is specified to the system.

2 State Machines

Dynamic systems can be described appropriately as state machines. This also holds for the multiagent systems we have just introduced. Therefore, statecharts are used quite often for the specification of dynamic or procedural aspects of software systems, also in industrial applications. In order to provide a better understanding of statecharts, we give their formal definition in the following.

2.1 Basic Components

Statecharts are a part of UML [10] and a well accepted means to specify dynamic behavior of software systems. The main concepts for statecharts is a state, which corresponds to an activity or behavior of an agent. Hence, statecharts also may be used in the context of *behavior-based programming*. They can be described in a rigorously formal manner, allowing flexible specification, implementation and analysis of multiagent systems [13].

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

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

E: a finite set of events,
V: a set of variables, where each variable $v \in V$ has an associated domain $dom(v)$, and
A: a finite set of actions, which sometimes may be composed from a sequence of simpler actions.

In statecharts, states are connected via transitions, that are annotated with conditions and actions. They are represented as rectangles with round corners and can be structured hierarchically. Following the lines of [10], we define this structure as follows. Each composite state $s \in S_{composite}$ has one *initial state* $\alpha(s)$. Each state $s \in S$ except s_0 *belongs to* a state $\beta(s)$. $\beta(s)$ is defined for all $s \in S \backslash \{s_0\}$ and it must hold $\beta(s) \in S_{composite} \cup S_{concurrent}$. If $\beta(s) \in S_{concurrent}$, then $s \in S_{composite}$, i.e., a concurrent state is never directly contained in another concurrent state.

A *transition* is a relation between two states indicating that an agent in the first state will enter the second state and perform specific *actions* when a specified *event* occurs and the specified conditions—called *guards*—are satisfied in addition. Transitions are drawn as arrows labeled with an *annotation* of the form $e[g]/a$ where $e \in E$, $g \in G$ (the set of guards, first-order formulæ, including equations with variables in V), and $a \in A$. Since states may be simple, composite or concurrent, the behavior of agents or their state machines, respectively, cannot be described by sequences of simple states (as for plain finite state machines), but of configurations.

Example 1. The state machine in Fig. 1 sketches the overall behavior of robotic soccer agents. Some details are not shown, which is indicated by the *hidden decomposition icon* ∘–∘. The machine contains the simple states Init and GetBall, the composite states Behave, Defend, Marking, and Attack, and the concurrent states HandleBall and Communicate. Obviously, the start state is Behave, whose initial state is Init. The three states Init, Attack and Defend belong to the main (start) state Behave. Its initial state Init permits a transition annotated with KickOff/Kick(100%) (with empty guard that corresponds to True) to Attack, if the agent has a KickOff from the center point.

2.2 Configurations and Steps

How can the behavior of state machines be described? For this, we now introduce the concepts of configuration and step. Configurations reflect the state the multiagent system is in. If a new composite state s is entered, then also the initial substates contained in s are entered. For this, we introduce the notion *completion* of a configuration, defining it in operational terms. This is more natural than giving a declarative definition (of a completed configuration), because not always all composite state machines are in their initial substates.

Definition 2 (Configuration). *A configuration c is a rooted tree of states, where the root node is the topmost initial state of the overall state machine. A*

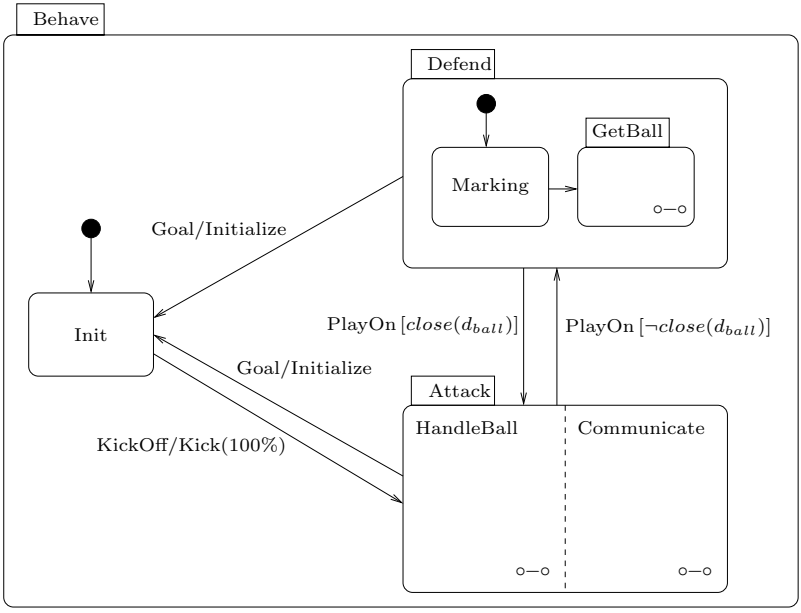

Fig. 1. State machine for the overall behavior.

configuration must be completed by applying the following procedure repeatedly as long as possible to leaf nodes:

1. *If there is a leaf node in c labeled with a composite state s, then* $\alpha(s)$ *is introduced as immediate successor of s.*
2. *If there is a leaf node in c labeled with a concurrent state s, then all (composite) states* s' *with* $\beta(s') = s$ *become successor nodes of s.*

In our context, state machines model the behavior of agents that act in their environment. The main effect of agents is that they interact with their environment. Therefore, state machines change the situation they are in, forming a trace, which is a sequence of situations. A *situation* σ is defined by mapping variables to values from given domains, characterizing the current world state, including the agent's configuration.

Agents perform steps from one configuration to another. For proper multiagent systems, we also must take into account that several agents or different components of one and the same agent may perform steps at the same time concurrently. Therefore, one may distinguish between micro- and macro-steps as in [12,13].

Definition 3 (Micro-step). *A micro-step from one configuration c of a state machine to a configuration* c' *by means of a transition t from state s to state* s' *with annotation* $e[g]/a$ *in the current situation—written* $c \rightarrow_t c'$*—is possible iff:*

1. *c contains a node labeled with s,*

2. *c' is identical with c except that s together with its subtree in c is replaced by the completion of s', and*
3. *the annotated event e and guard g hold in the actual situation.*

Fig. 2 shows some configurations of the state machine after several transitions. Since there is a transition from Init to Attack with annotation KickOff/Kick(100%) in the state machine, the step from the first to the second configuration shown in Fig. 2 is possible according to Def. 3. For this, the Init node is replaced by Attack. Since Attack is a concurrent state, the (composite) states HandleBall and Communicate belonging to Attack, become successor nodes of Attack. Again, these states have to be completed. This is indicated by triangles with the symbol ○—○ in it.

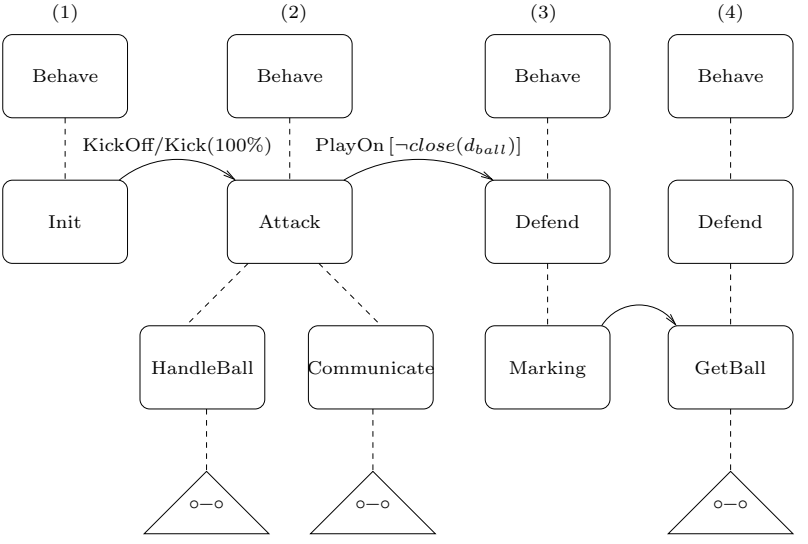

Fig. 2. Configuration after several transitions.

Definition 4 (Macro-step). *A macro-step from the configuration c to a configuration c' with given micro-steps is possible iff:*

1. *all micro-steps are possible in c,*
2. *all states $s_1, \ldots, s_l$ occur on different paths in the configuration tree c (i.e., all micro-steps are simultaneously applicable), and*
3. *c' is obtained by applying all given micro-steps to c.*

With macro-steps, we are able to model concurrent behavior. This is very important for the description of multiagent systems, since there we have several entities which interact concurrently. Concurrent states are the prerequisite for proper macro-steps, i.e., where several micro-steps are executed in parallel. In

order to be able to formalize systems of multiple agents, we require the set of variables V to contain variables for the actual event e—we assume that only one event occurs at a time (discrete time model)—and the configuration c of the whole multiagent system. Several agents can act in parallel, where a concurrent state (called region) is employed for each agent (see Fig. 3).

Fig. 3. Generic statechart for a system of multiple agents.

3 Specification of Multiagent Systems

Let us now consider the specification of multiagent systems by examples. The first and the second example (Sect. 3.1) is taken from the organizer domain, while the other one (Sect. 3.2) is from the robotic soccer domain. We will use the latter example in order to demonstrate our procedure for analyzing multiagent systems (Sect. 4).

3.1 Dynamic Behavior of an Agent and Synchronization

The meeting organizer agent, as introduced in Sect. 1.2, requires several input data, e.g. the subject, participants, starting time and room for the meeting. These requirements are related to each other and may be constrained by several conditions. The function of the meeting organizer agent is based on these data, therefore the agent changes its behavior dynamically, which corresponds to a state transition in UML statecharts.

Example 2. The main function of the meeting organizer agent obviously is the arrangement of a meeting. Let us introduce now the Organizing state, which represents its main function, with its substates. As depicted in Fig. 4 (left), the Organizing state of the meeting organizer agent is that a user selects the participants of the desired meeting at first. Secondly, room and time for the meeting is decided. The meeting organizer agent is able to make transitions into two exclusive substates: ArrangeTimeFirst substate and ArrageRoomFirst substate. Thus, a user can select which rule has priority to the other, in order to determine the room and the time of the meeting.

Example 3. Concurrent actions often involve synchronization, so that they can satisfy certain constraints. This can be conveniently expressed by synch states of UML statecharts. The ArrangeTimeFirst substate in Fig. 4 (left) has two concurrent actions, i.e. substates and their actions: one of them checks common

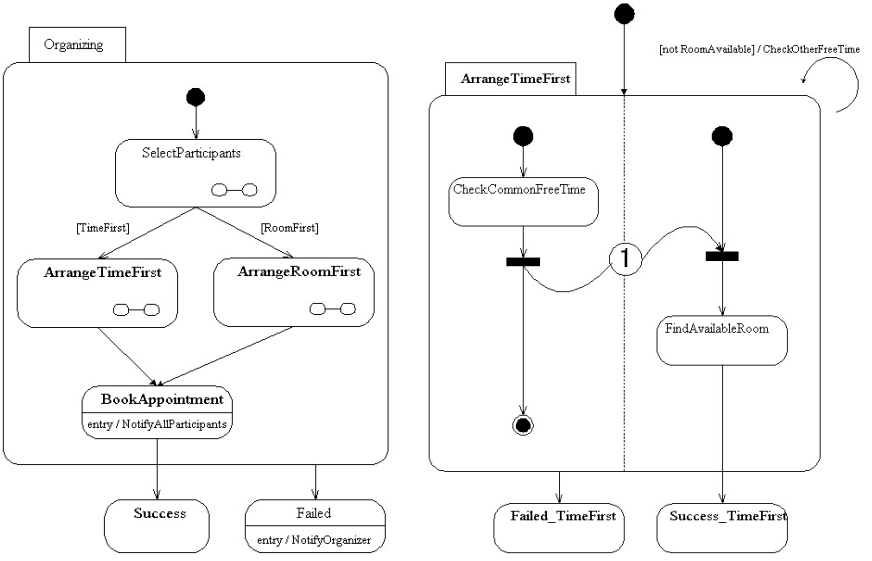

Fig. 4. State machine for Organizing substate and synchronized concurrent actions.

free time among the desired participants, and the other one tries to find an available room for the meeting at the time specified, see Fig. 4 (right). Here these two substates are mutually exclusive and concurrent. Moreover, the former procedure should be taken prior to the latter, and both substates need to be synchronized. If any room is available, the synch transition is fired, and the above procedure is taken again for a new combination of requirement data.

3.2 Obstacle Avoidance with Interrupts

Example 4. Let us consider now an example from the robotic soccer domain, describing the agent's behavior going to the ball. In principle, in this state the agent shall alternately turn its body towards the ball and go one step ahead. But if there is an obstacle in the way, then this procedure has to be interrupted, and the obstacle has to be avoided. The corresponding statechart is shown in Fig. 5. It makes use of a *history state*, shown as a circle with an H inside. Its semantics is that when returning into a state machine at a history state, then it is continued at the configuration that was valid when leaving it the last time.

This means, we have to memorize the actual configuration of the state machine with top state s containing the history state. For this, we introduce a variable h for each history state and each agent where $dom(h)$ is the set of all configurations of s restricted to H. For each transition in state s, the value of h has to be updated accordingly. In addition, when re-entering s, the state processing has to be continued with the stored configuration. Look at Fig. 5, where only the additional annotations for the treatment of the interrupt are shown (with abbreviated configuration names).

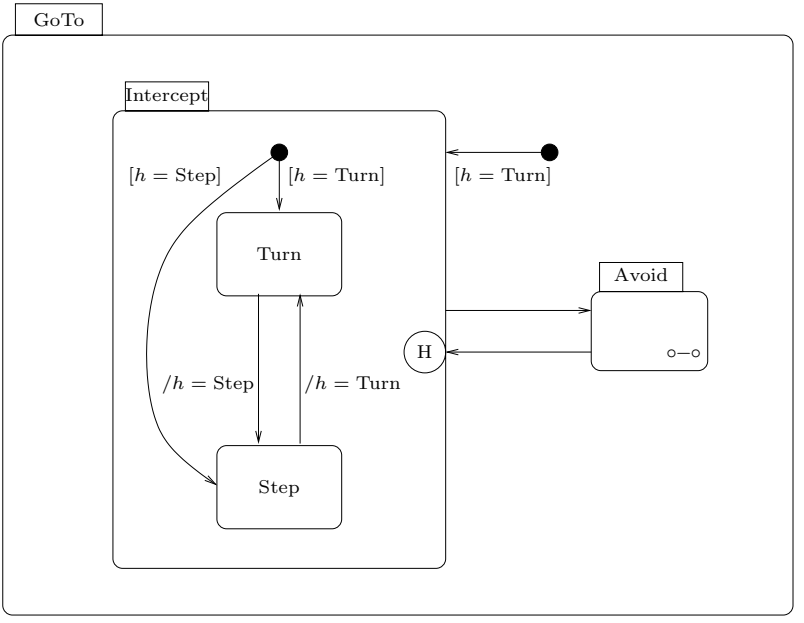

Fig. 5. Obstacle avoidance using history states.

Example 5. What are the different situations—given by assignments σ mapping each variable $v \in V$ into an element of its corresponding domain $dom(v)$—the agent may be in for this example? Situations change over time as an effect of actions the agents are performing. Since only the configuration c the agent is actually in and the history state h is interesting in this abstracted representation, we only have the following four different situations. Note that here we write configurations horizontally, in contrast to Fig. 2.

$$\sigma_1 : c = \text{Goto} \cdots \text{Intercept} \cdots \text{Turn}$$
$$\sigma_2 : c = \text{Goto} \cdots \text{Intercept} \cdots \text{Step}$$
$$\sigma_3 : c = \text{Goto} \cdots \text{Avoid} \ \wedge \ h = \text{Goto} \cdots \text{Intercept} \cdots \text{Turn}$$
$$\sigma_4 : c = \text{Goto} \cdots \text{Avoid} \ \wedge \ h = \text{Goto} \cdots \text{Intercept} \cdots \text{Step}$$

4 System Analysis by Model Checking

There are some properties of a system, that play a key role for the behavior of the system and its correctness. In multiagent systems, it is important to check system properties, e.g. whether from any state it is possible to reach a certain state, where a certain property must be satisfied. In industrial applications, especially for safety-critical applications, system behavior must be free from any failure.

4.1 Analysis with Kripke Structures

System specifications based on statecharts can be transformed into Kripke structures. This eventually allows us to reason over multiagent systems, by making use of temporal logics, e.g. Computation Tree Logic (CTL). For further information about Kripke structures and temporal logics, the reader is referred to [4].

Definition 5 (Kripke structure). *A* Kripke structure *consists of a finite set of states $\mathcal{S}$, a set $\mathcal{S}_0$ of initial states, a transition relation $R \subseteq \mathcal{S} \times \mathcal{S}$ that must be* serial *(i.e., for every state $s \in \mathcal{S}$ there is a state $s' \in \mathcal{S}$ with $R(s, s')$), and a function $L : \mathcal{S} \to 2^P$ that labels each state with the set of atomic propositions from the set P of all atomic propositions true in that state. In this context, an* atomic proposition *has the form $v = x$ with $v \in V$ and $x \in dom(v)$.*

Let us now describe, how a specification of a multiagent system by statecharts can be mapped into a Kripke structure, which yields the basis for the formal analysis of the original system. We can now define the following Kripke structure, which always exists for every given state machine. It is uniquely determined by the following definition:

1. The set of states $\mathcal{S}$ is the set of all valuations for V, which contains the actual system configuration c.
2. The set of initial states is a subset of all valuations which can be understood as situations σ satisfying the condition for the start situation, which is given by $c = s_0$, i.e., initially the state machine is in the (completed) start configuration.
3. The labeling function labels nodes with elements in Σ, the set of all situations.
4. Let $s, s' \in \mathcal{S}$, then $R(s, s')$ holds, if $L(s) \rightsquigarrow L(s')$, i.e., there is a macro-step leading from s to s' in effect. If, for some s, there is no s' such that $R(s, s')$, we add $R(s, s)$.

Example 6. Fig. 6 shows a Kripke structure modeling the state machine from Examples 4 and 5. One can see, that the agent steps and turns alternately (situations σ_1 and σ_2), unless it must do some obstacle avoidance (situations σ_3 and σ_4). Note that, since Fig. 5 in Sect. 3.2 is incomplete, the same is true for Fig. 6 in consequence.

4.2 Analyzing System Properties by Model Checking

Now we are ready for the formal verification of multiagent systems. By means of CTL, we are able to express safety-critical system properties. A transformation into Binary Decision Diagrams (BDDs) [4] can be done in order make formal analysis of the overall system as efficient as possible. For the sake of completeness, we introduce CTL briefly now. The syntax of CTL state (and path) formulæ is given by the following rules:

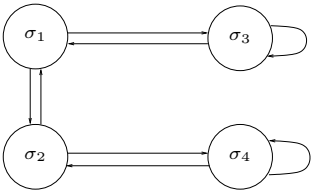

Fig. 6. Kripke structure for Example 5.

1. If p is an atomic proposition in P, then p is a state formula.
2. If f_0, f_1 and f_2 are state formulæ, then also $\neg f_0$, $f_1 \wedge f_2$ and $f_1 \vee f_2$.
3. If g is a path formula, then $\mathbf{E}\, g$ and $\mathbf{A}\, g$ are state formulæ.
4. If f_1 and f_2 are state formula, then $\mathbf{X}\, f_1$, $\mathbf{F}\, f_1$, $\mathbf{G}\, f_1$ and $f_1\, \mathbf{U}\, f_2$ are path formulæ.

The intended meaning of the temporal operators is as listed below. For more details, the reader again is referred to [4].

1. **E** ("for some computation path") denotes existential quantification.
2. **A** ("for all computation paths") denotes universal quantification.
3. **X** ("next time") requires that a property holds in the second state of the path.
4. **F** ("future") specifies that a property will hold at some state of the path.
5. **G** ("globally") specifies that a property holds at every state on the path.
6. **U** ("until") holds if there is a state on the path, where the second property holds, and on every preceding state on the path, the first property holds.

We can express planning tasks, but also more complex properties, e.g., that we can always achieve a certain goal. For instance, if we want to investigate whether we can reach a certain situation σ, i.e., there is a plan for achieving this goal (as in artificial intelligence planning [6]), then we just have to check $\mathbf{EF}\, \sigma$.

Example 7. We can now formulate temporal logic formulæ, checking e.g. that from any state it is possible to get to a state where obstacles are avoided, by the CTL formula $\mathbf{AG}(\mathbf{EF}\ c = \text{Goto} \cdots \text{Avoid})$. We can also express the fact that, by disregarding obstacle avoidance, the agent alternately steps and turns:

$$\begin{aligned}\mathbf{AG}\Big(\ & (\neg\, c = \text{Goto} \cdots \text{Avoid}\ \wedge \mathbf{AX} \neg\ c = \text{Goto} \cdots \text{Avoid}) \rightarrow \\ & (c = \text{Goto} \cdots \text{Intercept} \cdots \text{Turn} \rightarrow \mathbf{AX}\, c = \text{Goto} \cdots \text{Intercept} \cdots \text{Step}) \wedge \\ & (c = \text{Goto} \cdots \text{Intercept} \cdots \text{Step} \rightarrow \mathbf{AX}\, c = \text{Goto} \cdots \text{Intercept} \cdots \text{Turn})\Big)\end{aligned}$$

5 Related Works and Conclusions

We have presented a method for the graphical specification of multiagent systems. This method is applicable to different domains: robotic soccer and industrial applications, e.g. a concurrent work flow manager, as stated in this paper.

This approach is advantageous, because multiagent systems are then specified by means of only one type of diagram, namely UML statecharts (see also [1]). By this, more strict specification is possible and design of various systems is facilitated using one unified method.

We also demonstrated that such multiagent systems specification can be formally verified by means of model checking. There exists related work in various disciplines, e.g. automation technology [5]. Furthermore, in [3] abstract state machines are proposed for the specification of software systems, that are related to UML state machines; but for verification, classical theorem proving techniques are proposed, which might be difficult in practice for complex applications.

There are several software engineering approaches for the design and specification of multiagent systems. *Agent UML* [11] makes use of several diagrams, e.g., sequence diagram, communication diagram and interaction overview diagram, which are useful to specify especially the interaction of agents. [9] also employs UML and focuses on requirements. This designing methodology covers a wide range of the development cycle. In contrast to these methods, the emphasis of our approach is laid on the dynamic behavior of multiagent systems using one diagram, namely UML statecharts. In order to facilitate formal analysis and verification methods, we restrict ourselves to only one type of diagram.

Gaia [14] is another methodology for agent-oriented analysis and design. The key concepts in Gaia are roles, which have associated with them responsibilities, permissions, activities and protocols. A role can be viewed as an abstract description of an entity's expected function. However, this approach is neither intended for graphical specification nor for formal analysis as in our approach.

[2] presents an approach for model checking multiagent systems. It also makes use of temporal logics, but there the multiagent system is given by a BDI architecture, whereas we specify agents with statecharts. We think this is an advantage, because this allows us a quite high-level design of multiagent systems. [2] exploits the intrinsic modularity existing in multiagent systems. But in our approach, the modular, hierarchical structure of statecharts does not cause any problems, too. Note that the extensive use of composite states only leads to list-like configurations, and hence there is no combinatorial explosion of states in the Kripke structure after translation.

In this respect concurrent states are more problematic, because they lead to branching configuration trees. But this is a problem inherent to every multiagent system specification, if several agents act in parallel. Therefore, further work shall concentrate on how combinatorial explosion can be avoided in this case. One idea is to exploit dependencies among concurrent states (expressible by synchronization states). In addition, further work shall address even more collaborative behavior and communication in multiagent systems, so that integration of various systems is enabled in a unified process, where expressive specification and verification techniques as introduced in this paper play an important role.

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*, pages 11–18, Bologna, Italy, 2002. ACM Press. Volume 1.
2. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal of Logic an Computation*, 8(3):401–423, 1998.
3. E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer, Berlin, Heidelberg, New York, 2003.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, London, 1999.
5. T. Heverhagen. Verifikation von Funktionsbausteinadaptern durch Modelchecking. *Automatisierungstechnik*, 51(4):153–163, 2003.
6. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
7. J. Murray. Specifying agents with UML in robotic soccer. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems*, pages 51–52, Bologna, Italy, 2002. ACM Press. Volume 1.
8. J. Murray, O. Obst, and F. Stolzenburg. Towards a logical approach for soccer agents engineering. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, LNAI 2019, pages 199–208. Springer, Berlin, Heidelberg, New York, 2001.
9. J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The Tropos proposal. In M. Gogolla and C. Kobryn, editors, *Proceedings of the 4th International Conference on UML – The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, LNCS 2185. Springer, Berlin, Heidelberg, New York, 2001.
10. Object Management Group, Inc. *OMG Unified Modeling Language Specification*, March 2003. Version 1.5.
11. 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 (AOIS) at the 17th National Conference on Artificial Intelligence*, pages 3–17, 2000.
12. A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, LNCS 526, pages 244–264, 1991. Springer, Berlin, Heidelberg, New York.
13. F. Stolzenburg. Reasoning about cognitive robotics systems. In R. Moratz and B. Nebel, editors, *Themenkolloquium Kognitive Robotik und Raumrepräsentation des DFG-Schwerpunktprogramms Raumkognition*, Hamburg, 2001.
14. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.