Model Checking Hybrid Multiagent Systems for the RoboCup*

Ulrich Furbach[1], Jan Murray[1], Falk Schmidsberger[2], and Frieder Stolzenburg[2]

[1] Universität Koblenz-Landau, Artificial Intelligence Research Group, D-56070 Koblenz
{uli,murray}@uni-koblenz.de
[2] Hochschule Harz, Automation and Computer Sciences Department
D-38855 Wernigerode
{fschmidsberger,fstolzenburg}@hs-harz.de

Abstract. This paper shows how multiagent systems can be modeled by a combination of UML statecharts and hybrid automata. This allows formal system specification on different levels of abstraction on the one hand, and expressing real-time system behavior with continuous variables on the other hand. It is shown how multi-robot systems can be modeled by hybrid and hierarchical state machines and how model checking techniques for hybrid automata can be applied. An enhanced synchronization concept is introduced that allows synchronization taking time and avoids state explosion to a certain extent.

1 Multiagent Systems

Specifying behaviors for (physical) multiagent systems and multi-robot systems is a sophisticated and demanding task. Due to the high complexity of the interactions among agents and the dynamics of the environment the need for precise modeling arises. Since the behavior of agents usually can be understood as driven by external events and internal states, an obvious way of modeling multiagent systems is by state transition diagrams. Hierarchical state transition diagrams like statecharts are particularly well suited as they allow the specification of behaviors on different levels of abstraction [1].

One important aspect of physical agents and robots is that they interact with a (possibly simulated) physical environment. Such interactions typically consist of continuous actions (e.g. the movement of a robot) and perceptions like the power status of a battery. Classical state transition diagrams are not well suited for modeling this kind of interactions, as the transitions between states are discrete. However, continuous extensions to these formalisms have been proposed, e.g. hybrid automata [2].

Especially for agents employed in safety critical environments, e.g. in rescue scenarios, behavior specification has to be done very carefully in order to avoid side effects that may have unwanted or even disastrous consequences. One approach to realizing the required clarity of a specification is the use of formal design methods. Fortunately, many state transition diagram dialects like hybrid automata are equipped with a formal semantics that makes them accessible to formal validation of the modeled behavior. Thus it

* This research is supported by the grants *Fu 263/8* and *Sto 421/2* from the German research foundation *DFG* within the special priority program 1125 on *Cooperating Teams of Mobile Robots in Dynamic Environments*.

U. Visser et al. (Eds.): RoboCup 2007, LNAI 5001, pp. 262–269, 2008.

becomes possible to (semi-)automatically prove desirable features and the absence of unwanted properties in the specified behaviors, e.g. with model checking methods.

2 Hybrid Hierarchical State Machines

In this chapter, we present the combination of two concepts: hierarchical statecharts and hybrid automata. As a running example, we use a scenario from the RoboCup rescue simulation league, which is shortly described in the following subsection.

Rescue Scenario. In the RoboCup rescue simulation league [3], a large scale disaster is simulated. The simulator models part of a city after an earthquake. Buildings may be collapsed or on fire, and roads are partially or completely blocked. A team of heterogeneous agents consisting of police forces, ambulance teams, a fire brigade, and their respective headquarters is deployed. The agents have two main tasks, namely finding and rescuing buried civilians and extinguishing fires. An auxiliary task is clearing of blocked roads, such that agents can move smoothly. As their abilities enable each type of agent to solve only *one* kind of task (e.g. fire brigades cannot clear roads or rescue civilians), the need for coordination and synchronization among agents is obvious.

Consider the following simple scenario. If a fire breaks out somewhere, a fire brigade agent is ordered by its headquarters to extinguish the fire. The fire brigade moves to the fire and begins to put it out. If the agent runs out of water it has to refill its tank at a supply station and return to the fire to fulfill its task. Once the fire is extinguished, the fire brigade agent is idle again. An additional task the agent has to execute is to report any injured civilians it discovers. Part of this scenario is modeled in Fig. 1 with the help of a hierarchical hybrid automaton [4]. In addition to the fire brigade agent the model should include a fire station, fire, and civilians as part of the environment; all this will be explained in the next section (cf. Fig. 2).

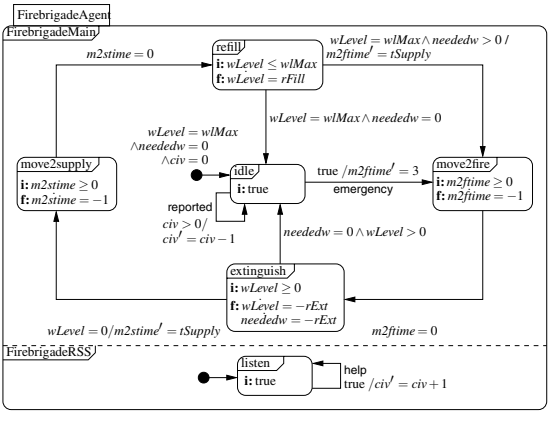

Fig. 1. A simple fire brigade agent

States are represented as rectangles with rounded corners and can be structured hierarchically. The specification of the fire brigade is a simple hierarchical chart, consisting of the main control structure (*FirebrigadeMain*) and a rescue sub-system (*FirebrigadeRSS*) which are supposed to run in parallel. The latter just records the detected civilians, which are not modeled in Fig. 1 (for this, see the sub-state *Civilians* in Fig. 2). *FirebrigadeMain* consists of five sub-states corresponding to movements (*move2fire*, *move2supply*), extinguishing (*extinguish*), refilling the tank (*refill*), and an idle state (*idle*). The agent can report the discovered civilians when it is in its idle state. Details from this figure will be explained in the course of this section.

Obviously, even in this simple case with few components and a deterministic environment it is difficult to see if the agent behaves correctly. Important questions like "does the fire brigade try to extinguish without water?" or "will every discovered civilian (and *only* those) be reported eventually?" depend on the interaction of all components and cannot be answered without an analysis of the whole system.

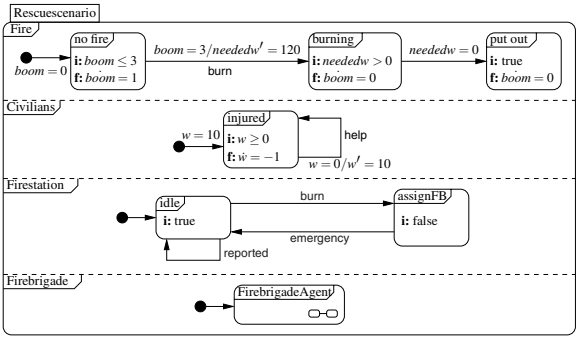

Fig. 2. A simple scenario from the RoboCup rescue simulation. The state *FirebrigadeAgent* corresponds to Fig. 1. The icon ⬭–⬭ hints at the hidden sub states.

State Hierarchies and Transitions. Statecharts are a part of UML [5,6] and a well accepted means to specify dynamic behavior of software systems. The main concept for statecharts is a state, which corresponds to an activity or behavior of a robot agent. Statecharts can be described in a rigorously formal manner [7,8], allowing flexible specification, implementation and analysis of multiagent systems [1] which is required for robot behavior engineering and modeling and simulating complex robots.

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

S: a finite set of states, which is partitioned into three disjoint sets: S_{simple}, S_{comp}, *and* S_{conc} *— called simple, composite and concurrent states, containing one designated* start state $s_0 \in S_{comp} \cup S_{conc}$, *and*
X: a finite set of (real-numbered) variables.

In our running example, *idle*, *extinguish* or *listen* are simple states, and *FirebrigadeAgent* is a concurrent state and *FirebrigadeMain* and *FirebrigadeRSS* are composite states, called regions in this case, which are separated by a dashed line. *m2ftime* and *wLevel* are examples for real valued variables.

In statecharts, states are connected via *transitions* in $T \subseteq S \times S$, indicating that an agent in the first state will enter the second state. Transitions are drawn as arrows labeled with jump conditions over the variables in X together with actions. For example, the transition from *idle* to itself is labeled with $civ > 0/civ' = civ - 1$, with the meaning: if the value of *civ* is greater 0, the action $civ' = civ - 1$ is executed while performing the transition, i.e., the number of civilians that are found but not reported is decreased.

The label reported at the same transition is used for synchronizing the transition with another automaton working in parallel, namely the one for *Firestation* (see Fig. 2). It is only legal for the combined system if both automata take the transition labeled reported at the same time (see [2] for details). In principle, the explicit use of events and actions as in UML statecharts is not needed, as both can be expressed with the help

of variables. For example the occurrence of an external event can be represented by changing the value of the corresponding variable from 0 to 1.

Since hybrid automata [2] are similar to statecharts, it makes sense to combine the advantages of both models. Statecharts have the clear advantage of allowing hierarchical specification on several levels of abstraction, while hybrid automata enable the introduction of continuous variables and flow conditions. This extension of statecharts is done by the subsequent definition. Hybrid automata are widely used for the specification of embedded systems. By reachability analyses, diagnosis tasks can be solved. We will come back to this in Sect. 4.

Definition 2 (jump conditions, flows and invariants). *In addition to the variables in X, we introduce new variables $\dot{x}$ (first derivatives during continuous change) and x' (values at the conclusion of discrete change) for each $x \in X$, calling the corresponding variable sets $\dot{X}$ and X', respectively. Then, each transition in T may be labeled by a* jump condition*, that is a predicate whose free variables are from $X \cup X'$, which can be split into activation condition and effect. In addition, each state $s \in S$ is labeled with a* flow condition *(**f:**), whose free variables are from $X \cup \dot{X}$, and an* invariant *(**i:**), whose free variables are from X. Flow conditions may be empty and hence omitted, if nothing changes continously in the respective state.*

In our example we use the dotted variable $\dot{wLevel}$ to denote the change of the water level in the state *refill*. A transition from this state to the state *move2fire* is performed, if the water level reached the maximum ($wLevel = wlMax$) and water is needed ($neededw > 0$). During the transition the action $m2ftime' = tSupply$ is executed.

We will restrict our attention to linear conditions, i.e. linear equalities and inequalities among either ordinary variables in $X \cup X'$ or their first derivatives $\dot{X}$, because only then an exact reachability analysis (needed for model checking) is feasible [2]. Following the lines of [5,6], UML statecharts have a hierarchical structure which can easily be represented as a tree of states. Here, regions with cardinality greater than one must be treated as multiple composite states, which are distinguished by different indices. The behavior of the overall state machine can be described by a sequence of state tree configurations, called micro-steps. For this, the interested reader is referred to [7,9].

3 Synchronization and Cooperation

The overall performance of programmed multiagent systems heavily depends on how cooperative agents behave. Cooperation and coordination of agents can be achieved by synchronization. Hence, it is essential to implement synchronization effectively. Synchronization means that several actions must start or happen at the same time. In the rescue scenario (see Sect. 2), transition labels serve as triggers for synchronization in the formalism of hybrid automata, e.g., if an injured civilian cries for help, then the listening fire fighter hears this. However, if more complicated coordination and cooperation among agents has to be expressed, then this simple concept of synchronization may not suffice. In the following, we will therefore introduce an enhanced concept of synchronization (see [4]), which we motivate with an example from robotic soccer.

An Example of Coordination in Robotic Soccer. Since (robotic) soccer is a team sport, cooperation of agents is essential.

At best, exactly one player should go to the ball, while the others try to position themselves as good as possible. Fig. 3 shows the statechart for two players trying a coordinated behavior of going to the ball. To realize this behavior, the positions of two players (**p1, p2**), the ball (**bR**), a (stationary) opponent (**PO**) and the opponent goal **POG** are modeled. Additionally the local estimates of the ball position **b**, the player **p** and its teammate **pT** are given for each player. Finally there is the players' measurement error *ME*, the range *DHB*, within which a player has the ball, and some scaling factors $F01$–$F04$. Constant names start with capital letters, variables with lower case letters.

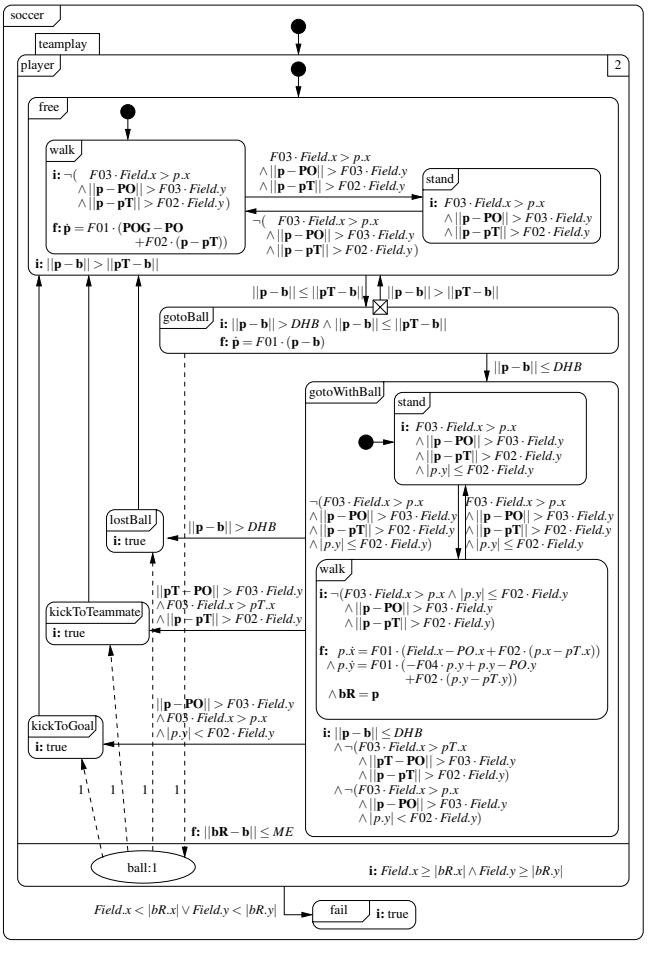

Fig. 3. Robotic soccer example

In this example, coordination is really important. In contrast to simple synchronization mechanisms, coordination may take some time. The time between deciding to go to the ball and actually reaching it will be almost always greater than zero. Thus, we must be able to distinguish between the allocation and the occupation of a resource (e.g. the ball) in our specification formalism. In addition, since coordination may take some time, we associate the new synchronization method with states and not with transitions. All this is comprised in the concept of *timed synchronization* introduced next.

Timed Synchronization. Usually the so-called *synchrony hypothesis* is adopted for state machines, assuming that the system is infinitely faster than the environment and thus the response to an external stimulus (event) is always generated in the same step

that the stimulus is introduced. However in practice, synchronization and coordination of actions cannot be done in zero time. In UML 1.5 [5], synchronization is present, but assumed to take zero time. In UML 2.0 [6] there does not seem to be a special synchronization mechanism available any longer except by join and fork transitions. Hence, it seems to be really worthwhile considering synchronization and coordination in more detail. For this, we will introduce synchronization points which are associated with states, i.e. activities that last a certain time, and not with transitions (as in UML 1.5), because the transition from one state to another takes zero time according to the synchrony hypothesis.

Definition 3 (synchronization points). *A* synchronization point *(represented as oval) allows the coordinated treatment of common resources. It can be identified by special synchronization variables* $x \in X_{synch} \subseteq X$ *with a given maximal* capacity $C(x) > 0$. *Each such point may be connected with several states. We distinguish two relations:* $R_+ \subseteq S \times X_{synch}$ *and* $R_- \subseteq X_{synch} \times S$, *both represented by dashed arrows in the respective direction. Further, each connection in* $R_+ \cup R_-$ *is annotated with a number* m *with* $0 < m \leq C(x)$.

As just said, according to the previous definition, synchronization is connected to states and not to transitions as in UML 1.5. In consequence, it is now possible that synchronization may take some time as desired. The process of synchronization starts when a state s connected to a synchronization variable x is entered, and it ends only after some time when s is exited. Therefore, we distinguish the allocation of (added or subtracted) resources and their (later) actual occupation.

In Fig. 3, coordination is achieved by the synchronization variable *ball* introduced in the concurrent state *teamplay*. It has capacity 1, as there is only one ball in a soccer game. The *gotoBall* state is positively connected to it, while the states *kickToGoal*, *kickToTeammate*, and *lostBall* are negatively connected to it. This means, that the ball resource is allocated during the *gotoBall* activity and deallocated after a kick. The transition marked with a crossed box indicates a failed synchronization.

4 Model Checking

As we already mentioned, hybrid automata are equipped with a formal semantics, which makes it possible to apply formal methods in order to prove certain properties of the specified systems, e.g. by model checking. However, in the context of hybrid automata the term *model checking* usually refers to *reachability* testing, i.e. the question whether some (unwanted) state is reachable from the initial configuration of the specified system. To this end, all states that can be reached by a discrete transition or evolving the continuous variables according to a flow condition are repeatedly added to the current configuration until a fixpoint R is reached. Then it can be tested, if unwanted states are reachable simply by intersecting the sets of reachable and unwanted states.

For the behavior specification shown in Figs. 1 and 2 we conducted several experiments with the standard model checkers HyTech [10]. Both model checkers are implemented for the analysis of linear hybrid automata. They take textual representations of hybrid automata like the one in Fig. 4 as input and perform reachability tests on the

state space of the resulting product automaton. This is usually done by first computing all states reachable from the initial configuration, and then checking the resulting set for the needed properties. In the remainder of this section, we present some exemplary model checking tasks for the rescue scenario.

Is it possible to extinguish the fire? When the state of the automaton modeling the fire changes from *no fire* to *burning*, the variable *neededw* stores the amount of water needed for putting out the fire (*neededw* $= 120$ in the beginning). When the fire is put out, i.e. *neededw* $= 0$, the automaton enters the state *put out*. Thus the fire can be extinguished, iff there is a reachable configuration c_{out} where fire is in the state *put out*. It is easy to see from the specification, that this is indeed the case, as *neededw* is only decreased after the initial setting, and so the transition from *burning* to *put out* is eventually forced. With the help of HyTech's trace generation ability it is quite easy to solve the additional task of comparing different strategies, e.g. for refilling the water tanks. To this end, traces to c_{out} generated using the different strategies are compared. A shorter trace (wrt. time units) corresponds to a faster solving of the extinguishing task.

Does the agent try to extinguish with an empty water tank? The fact that the firebrigade agent tries to put out the fire without water corresponds to the simple state *extinguish* being active while *wLevel* < 0. Note that we must not test for *wLevel* ≤ 0, as the state *extinguish* is only left when the water level is zero, so including a check for equality leads to false results. Fig. 4 shows how to check this property with HyTech. The set of reachable states is collected in the variable `init_reach` (l. 8), and `ext_error` is assigned the set of illegal states (l. 9), i.e., all states where extinguish is active and the water level is below zero. Lines 10–12 finally show the actual test. If the intersection of reachable and illegal states is not empty (l. 10), an error message is printed (l. 11).

Does the agent report all discovered civilians? This question contains two properties to be checked: (a) all discovered civilians are reported eventually, and (b) the agent does not report more civilians than he found. The discovery of a civilian is modeled by increasing the value of the variable *civ* by one. For each reported civilian one is subtracted from *civ*. From this it follows, that (b) holds, iff no configuration is reachable, where *civ* < 0. To show (a), one has to ensure that from all configurations with *civ* > 0 a configuration with *civ* $= 0$ will be reached eventually. Testing these properties with HyTech reveals that (b) holds in the specification, i.e. for all reachable states *civ* ≥ 0. However, the analysis also yields that (a) does not hold. As we stated earlier the fire fighter agent should report civilans when he is in the *idle* state. But as the invariant in this state (true) is never violated, the agent is not forced to take the self transition labeled reported, which corresponds to reporting a civilian. Thus, there is a legal run of the system, where no civilian is reported at all.

```
 1 automaton Civilian
 2   synclabs: help;
 3   initially injured & w = -10;
 4   loc injured:
 5     while w<=0 wait {}
 6     when w=0 sync help do {w' = -10} goto injured;
 7 end

 8 init_reach := reach forward from init endreach;
 9 ext_error := loc[FirebrigadeMain] = extinguish & wLevel < 0;
10 if not empty(init_reach & ext_error)
11   then prints "Error: Tank empty!";
12 endif;
```

Fig. 4. HyTech code for the civilian automaton from Fig. 2 (ll. 1–7) and analysis commands

It should be remarked that synchronization points help us to reduce complexity. In order to see this, let us consider a composite state with cardinality *m* containing *k* (simple)

states. One of them, say s, is connected to a synchronization point with capacity C. Then there are k^m different configurations, i.e. exponentially. Since at most C agents can be in s, only $\sum_{l=0}^{C} \binom{m}{l}(k-1)^{m-l}$ configurations have to be considered. This is polynomial for $k = 2$. A naïve flattening of the example in Fig. 3 e.g. leads to $8 \cdot 8 + 1 = 65$ configurations, whereas taking synchronization into account leads to only $2 \cdot 2 + 2 \cdot 2 \cdot 6 + 1 = 29$ configuration states. A translator that automatically converts hybrid hierarchical statecharts into simple flat hybrid automata is currently implemented [11].

5 Conclusions

In this paper we demonstrated the use of hybrid hierarchical state machines for the specification of multiagent systems. We presented two application scenarios from the RoboCup, one from the rescue simulation and one from robotic soccer, and we demonstrated that state-of-the-art model checkers for hybrid automata can be used for proving properties of the specified systems. We exemplified this especially with an example from the RoboCup rescue scenario. Model checking, i.e. reachability analysis helps us finding out possible paths, which could help in the pre-computation of multiagent system implementations. This point will be subject of future work as well as studies of whether the procedure scales up to more complex scenarios.

References

1. Murray, J.: Specifying agent behaviors with UML statecharts and StatEdit. In: Polani, D., Browning, B., Bonarini, A., Yoshida, K. (eds.) RoboCup 2003. LNCS (LNAI), vol. 3020, pp. 145–156. Springer, Heidelberg (2004)
2. Henzinger, T.: The theory of hybrid automata. In: Proceedings of the 11th Annual Symposium on Logic in Computer Science, pp. 278–292. IEEE Computer Society Press (1996)
3. Tadokoro, S., et al.: The RoboCup-Rescue project: A robotic approach to the disaster mitigation problem. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA 2000), pp. 4089–4104 (2000)
4. Murray, J., Stolzenburg, F.: Hybrid state machines with timed synchronization for multirobot system specification. In: Bento, C., et al. (eds.) Proceedings of 12th Portuguese Conference on Artificial Intelligence, pp. 236–241. IEEE Inc. (2005)
5. Object Management Group, Inc.: UML Specification, Version 1.5 (March 2003)
6. Object Management Group, Inc.: UML 2.0 Superstructure Specification (October 2004)
7. Arai, T., Stolzenburg, F.: Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In: Proceedings of 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems, pp. 11–18. ACM Press (2002)
8. Pnueli, A., Shalev, M.: What is in a step: On the semantics of statecharts. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 244–264. Springer, Heidelberg (1991)
9. Furbach, U., Murray, J., Schmidsberger, F., Stolzenburg, F.: Hybrid multiagent systems with timed synchronization – specification and model checking. In: Dastani, M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 205–220. Springer, Heidelberg (2008)
10. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: The Next Generation. In: IEEE Real-Time Systems Symposium, pp. 56–65 (1995)
11. Ruh, F.: A translator for cooperative strategies of mobile agents for four-legged robots. Master thesis, Dept. of Automation and Computer Sciences, Hochschule Harz (2007)