Hybrid Multiagent Systems with Timed Synchronization- Specification and Model Checking

Ulrich Furbach
Jan Murray
Falk Schmidsberger
Frieder Stolzenburg

Nr. 14/2007

Arbeitsberichte aus dem Fachbereich Informatik

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Paulus

Kontaktdaten der Verfasser

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg
Institut für Informatik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: uli@uni-koblenz.de, murray@uni-koblenz.de, fschmidsberger@hs-harz.de, fstolzenburg@hs-harz.de

Hybrid Multiagent Systems with Timed Synchronization – Specification and Model Checking,
Fachbereich Informatik, Nr. 14/2007

Hybrid Multiagent Systems with Timed Synchronization – Specification and Model Checking

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 not only shown how multi-robot systems can be modeled by a combination of hybrid automata and hierarchical state machines, but also 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 [6]. They can directly be used as executable specifications for programming multiagent systems [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 [4].

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 result in unwanted behaviors or even have 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 becomes possible to (semi-)automatically prove desirable features and the absence of unwanted properties in the specified behaviors, e.g. with the help of 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.

2.1 Rescue Scenario

In the RoboCup Rescue Simulation League [13], 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. In this scenario, 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 buried roads, so 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 [7]. 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).

States are represented as rectangles with rounded corners and can be structured hierarchically. The specification of the fire brigade is a simple hierarchical chart (see Fig. 1), 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; it should be obvious already in this stage, that 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?

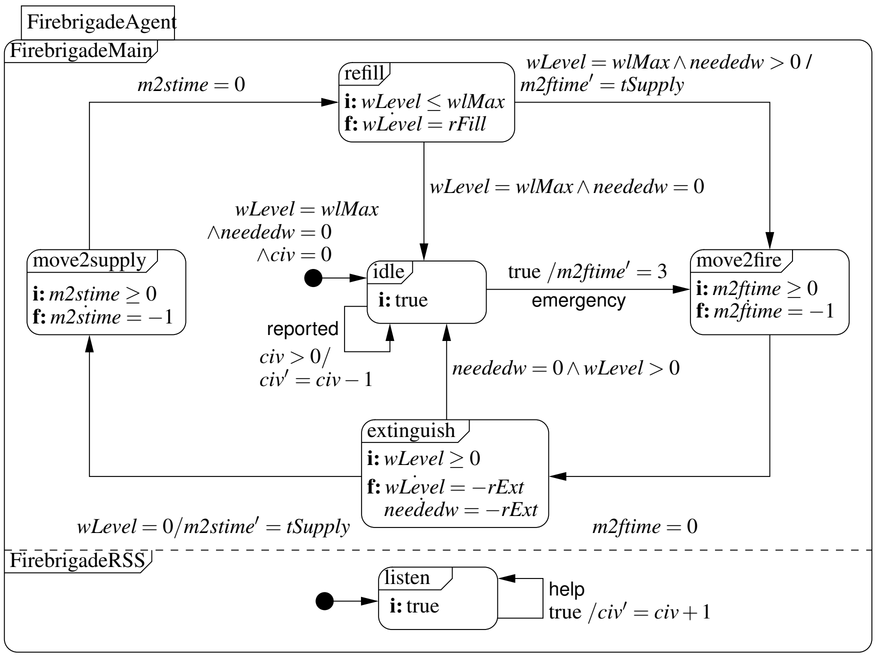

Fig. 1. A simple fire brigade agent.

– 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.

2.2 State Hierarchies and Transitions

Let us now define the notation used so far more formally. Statecharts are a part of the unified modeling language UML [9, 10] 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. They can be described in a rigorously formal manner [1, 11], allowing flexible specification, implementation and analysis of multiagent systems [1, 6, 12] 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 in this case. 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 [4] 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 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 $X \cup X'$. 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, 4]. Let us now have a closer look at states. Following the lines of [9, 10], we define the hierarchical structure of statecharts as follows.

Definition 3 (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. Furthermore, each state $s \in S \backslash \{s_0\}$* belongs to *exactly*

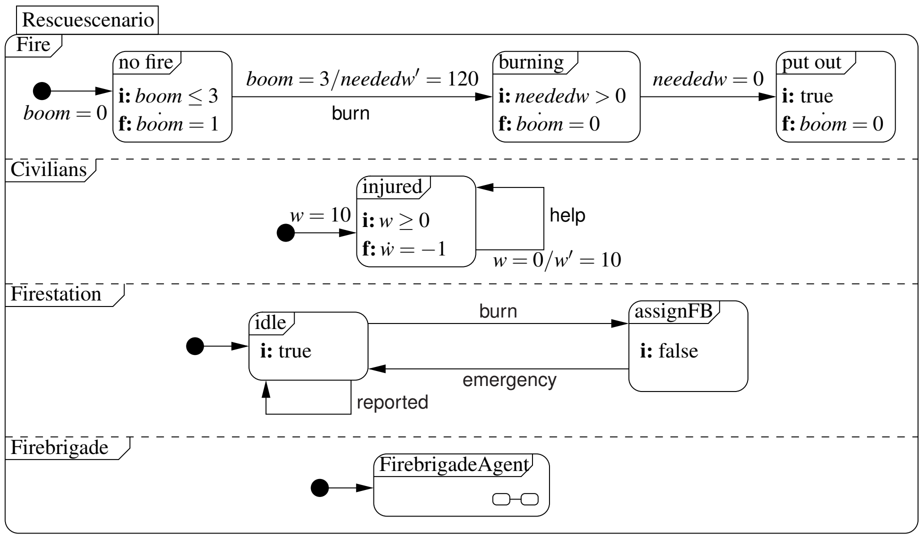

Fig. 2. A simple scenario from the RoboCup rescue simulation. The state *FirebrigadeAgent* corresponds to the one shown in Fig. 1. The icon ⊂⊃–⊂⊃ hints at the hidden sub states.

one state $\beta(s)$ different from s. It must hold $\beta(s) \in S_{comp} \cup S_{conc}$. If $\beta(s) \in S_{conc}$, then $s \in S_{comp}$, which implies that a concurrent state must not be directly contained in another concurrent state, as they could be merged into a single concurrent state in this case. s is called region *of $\beta(s)$ then and may have a* cardinality *greater than one, which is expressed by a cardinality marker in the upper right corner of a region; if the cardinality is one, the marker may be omitted. We assume that transitions keep to the hierarchy, i.e., if sTs' holds, then $\beta(s) = \beta(s')$.*

In Fig. 1 we see that the start state s_0 is *FirebrigadeAgent*, a concurrent state. It represents the multiagent system, consisting of an agent *FirebrigadeMain* and *FirebrigadeRSS*. Both all realized as regions, which are separated by dashed lines (in the case of heterogenous agents), and each has cardinality one. The entire rescue scenario, which we will also use for model checking later on is depicted in Fig. 2; besides the fire brigade we additionally have concurrent regions with states for *Fire*, *Civilians* and *Firestation*.

2.3 State Trees and Configurations

The function β (see Def. 3) naturally induces a state tree with s_0 as root. This is shown for the running example in Fig. 3. Here, regions with cardinality greater than one must be treated as multiple composite states, which are distinguished by different indices. However, while processing, each region or composite state of the state machine contains only one active state. These states also form a tree, called configuration. A configuration of the given state machine, is indicated by the thick lines in Fig. 3.

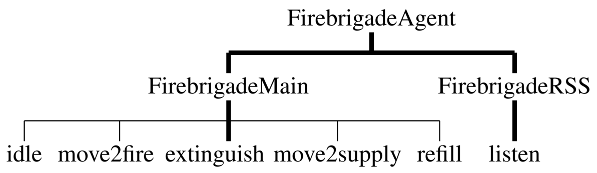

Fig. 3. State hierarchy and configuration tree (thick lines).

Definition 4 (configuration). *A* configuration *c is a rooted tree of states, where the root node is 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.

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, because it may take non-zero time. In the following, we will therefore introduce an enhanced concept of synchronization (see [8]), which we motivate with an example from the robotic soccer domain.

3.1 An Example of Coordination in Robotic Soccer

Since (robotic) soccer is a team sport, cooperation of agents is essential. Clearly, it is not a good idea that all players try to get the ball at the same time. At best, exactly one player goes to the ball, while the others try to position themselves as good as possible on the pitch.

Fig. 4 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. The positions are described as two-dimensional vectors $\mathbf{v} = \binom{x}{y}$. Components are accessed via the point notation, e.g. $v.x$. Constant names start with capital letters, variables with lower case letters.

There are variables for the global, real ball position **bR** (initially $\binom{80}{60}$), the local ball position **b** measured by each player, global positions of the players 1 and 2 (initial values $\mathbf{p1} = \binom{0}{60}$, $\mathbf{p2} = \binom{0}{-60}$), the local position of the player **p** and his teammate **pT**

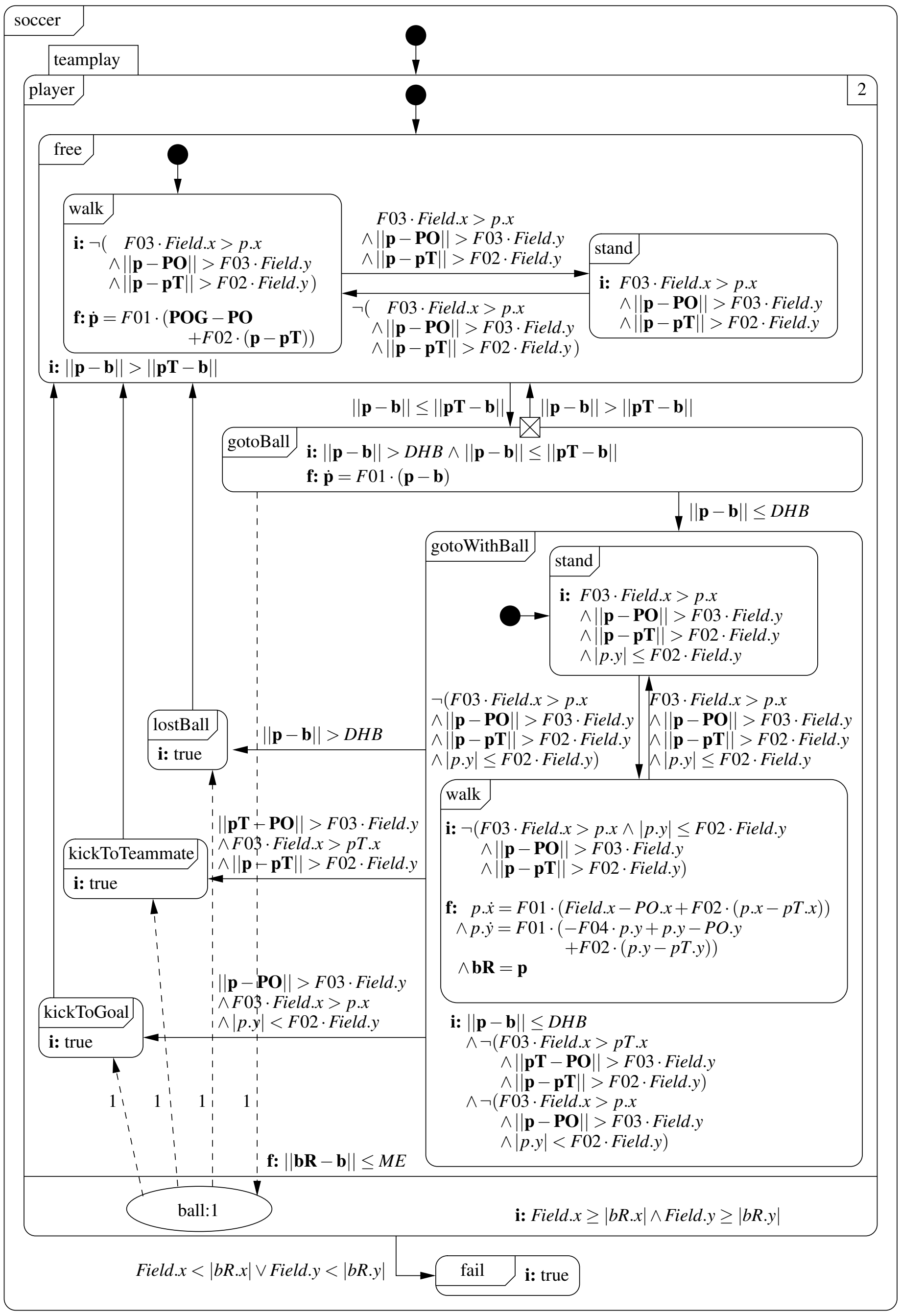
soccer
teamplay
player
2
free
walk
i: $\neg(\ F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y)$
f: $\dot{\mathbf{p}} = F01 \cdot (\mathbf{POG}-\mathbf{PO} + F02 \cdot (\mathbf{p}-\mathbf{pT}))$
$F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y$
$\neg(\ F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y)$
stand
i: $F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y$
i: $\|\mathbf{p}-\mathbf{b}\| > \|\mathbf{pT}-\mathbf{b}\|$
$\|\mathbf{p}-\mathbf{b}\| \leq \|\mathbf{pT}-\mathbf{b}\|$
$\|\mathbf{p}-\mathbf{b}\| > \|\mathbf{pT}-\mathbf{b}\|$
gotoBall
i: $\|\mathbf{p}-\mathbf{b}\| > DHB \wedge \|\mathbf{p}-\mathbf{b}\| \leq \|\mathbf{pT}-\mathbf{b}\|$
f: $\dot{\mathbf{p}} = F01 \cdot (\mathbf{p}-\mathbf{b})$
$\|\mathbf{p}-\mathbf{b}\| \leq DHB$
gotoWithBall
stand
i: $F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y \wedge |p.y| \leq F02 \cdot Field.y$
$\neg(F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y \wedge |p.y| \leq F02 \cdot Field.y)$
$F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y \wedge |p.y| \leq F02 \cdot Field.y$
walk
i: $\neg(F03 \cdot Field.x > p.x \wedge |p.y| \leq F02 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y)$
f: $p.\dot{x} = F01 \cdot (Field.x - PO.x + F02 \cdot (p.x - pT.x)) \wedge p.\dot{y} = F01 \cdot (-F04 \cdot p.y + p.y - PO.y + F02 \cdot (p.y - pT.y)) \wedge \mathbf{bR} = \mathbf{p}$
i: $\|\mathbf{p}-\mathbf{b}\| \leq DHB \wedge \neg(F03 \cdot Field.x > pT.x \wedge \|\mathbf{pT}-\mathbf{PO}\| > F03 \cdot Field.y \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y) \wedge \neg(F03 \cdot Field.x > p.x \wedge \|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge |p.y| < F02 \cdot Field.y)$
lostBall
i: true
$\|\mathbf{p}-\mathbf{b}\| > DHB$
kickToTeammate
i: true
$\|\mathbf{pT}-\mathbf{PO}\| > F03 \cdot Field.y \wedge F03 \cdot Field.x > pT.x \wedge \|\mathbf{p}-\mathbf{pT}\| > F02 \cdot Field.y$
kickToGoal
i: true
$\|\mathbf{p}-\mathbf{PO}\| > F03 \cdot Field.y \wedge F03 \cdot Field.x > p.x \wedge |p.y| < F02 \cdot Field.y$
1
1
1
1
f: $\|\mathbf{bR}-\mathbf{b}\| \leq ME$
ball:1
i: $Field.x \geq |bR.x| \wedge Field.y \geq |bR.y|$
$Field.x < |bR.x| \vee Field.y < |bR.y|$
fail
i: true

Fig. 4. Robotic soccer example.

and some constants for the global position of the (stationary) opponent $\mathbf{PO} = \binom{110}{-30}$, and the opponent goal $\mathbf{POG} = \binom{Field.x}{0}$, where **Field** denotes the field size. The field reaches from $-$**Field** to $+$**Field**. Further there is the measurement error $ME = 2$ of the players, the range within a player has the ball $DHB = 5$ and some scale factors $F01 = 0.1$, $F02 = 0.5$, $F03 = 0.3$, and $F04 = 0.6$. To access a local value, the path over the states to the value is used. For instance, the local position of player 1 is soccer.teamplay.player1.**p** with the initial value **p1** and the local position of his teammate is soccer.teamplay.player1.**pT** with the initial value **p2**. The composite state *soccer* contains the concurrent state *teamplay* as initial state and the simple state *fail*. There is only one transition from *teamplay* to *fail*, and *fail* can only entered, if the invariant of *teamplay* is false and the guard of the transition is true. In this case, the ball has to be out of the bounds of the field. Note that the synchronization variable *ball* and the invariant beside it belongs to *teamplay*.

The behavior of the two players is modeled in the regions *player* inside of *teamplay*, which is a concurrent state with two regions: one for each of the two players. But since both players obey in principle the same specification, i.e., we have a homogeneous agent system (expressed by a cardinality marker in the upper right corner). The initial state of *player* is *free* (running freely) with the following behavior. The player moves to an optimal position related to **POG**, **PO** and **pT** (state *walk*). If he is in an optimal position, he waits for the ball passed from the teammate (state *stand*). Otherwise he moves on. If the player is closer to the ball than his teammate, his state is changing from *free* to *gotoBall*. The flow condition inside *gotoBall* is modeling the movement of the player to reach the ball position. If his teammate becomes now closer to the ball, the player will fall back to the state *free*. Otherwise, if his distance to the ball becomes less than *DHB*, his state changes to *gotoWithBall*.

Inside *gotoWithBall*, the following behavior is modeled. The player dribbles the ball to an optimal position related to **PO**, **pT** and the center in front of the opponent goal (state *walk*). If he is in an optimal position, he waits (state *stand*) with the ball to pass to the teammate or to kick to the goal, otherwise he moves on. There are 3 transitions out of *gotoWithBall*. If the distance to the ball becomes greater than *DHB*, the player loses the ball (state *lostBall*) and changes further to *free*. If **p**, **PO** and **pT** are optimal for a pass, the player will kick the ball to his teammate (state *kickToTeamMate*) and changes to *free*.If **p** and **PO** are optimal in front of the opponent goal, the player will kick the ball to the opponent goal (state *kickToGoal*) and afterwards he changes to free. The flow conditions of the last three states are omitted for a better clarity of the figure.

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.

3.2 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 [9], synchronization is present, but assumed to take zero time. In UML 2.0 [10], 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 5 (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 by additional variables x_+ and x_- (used during the allocation phase) in each synchronization point. Hence, for each $x \in X_{synch}$, x_+ and x_- must be added to X.

In the following, we write $\alpha^n(s)$ or $\beta^n(s)$ for the n-fold application of α or β to s, especially $\alpha^0(s) = \beta^0(s) = s$. Let us now have a closer look at variables. 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. All synchronization variables and their relatives are global in principle. Nevertheless, we associate synchronization points identified by the variable x with the state $\gamma(x)$ where it is declared; $\gamma(x)$ must be a concurrent state in this case. Therefore we assume, that for all states s connected to x, i.e. sR_+x or xR_-s, it must hold $\beta^n(s) = \gamma(x)$ for some $n \geq 0$, and all s' between s and $\gamma(x)$ in the state tree must be composite states.

Definition 6 (transition types). *Let x be a synchronization variable introduced at $\gamma(x)$ and s be a state connected with x. Then, s_1Ts_2 is called* incoming transition *for s iff $\alpha^n(s_2) = s$ for some $n \geq 0$. It is called* initializing*, if it is an incoming transition with $\alpha^n(s_2) = \gamma(x)$ for some $n \geq 0$. s_1Ts_2 is called an* outgoing transition *for s iff $s_1 = \beta^n(s)$ for some $n \geq 0$, where s_1 occurs in the actual configuration tree and x is valid in s. It* *is called* successful*, if it is an outgoing transition with $s = s_1$ and not marked with a crossed box* ⊠*; otherwise, it is called* failed*.*

Note that outgoing transitions cannot be characterized statically by the state hierarchy, but by the actual configuration tree. For the ease of presentation, we assume that there is a special *start transition* leading to s_0, annotated with a given *initial condition* of the whole state machine. For this, an artificial new start state may be introduced.

Definition 7 (synchronization constraints). *Synchronization points impose additional constraints to the transitions that are incident with states s, the synchronization variables x are connected to.*

1. *If sR_+x with annotation m, then*
   (a) *$x + x_+ + m \leq C(x)$ and $x'_+ = x_+ + m$ are added to all not initializing incoming transitions,*
   (b) *$x' = 0, x'_+ = 0, x'_- = 0$ are added to all initializing incoming transitions,*
   (c) *$x'_+ = x_+ - m$ is added to all outgoing transitions, and*
   (d) *$x' = x + m$ is added to all successful outgoing transitions supplementarily.*
2. *If xR_-s with annotation m, then*
   (a) *$x - (x_- + m) \geq 0$ and $x'_- = x_- + m$ are added to all not initializing incoming transitions,*
   (b) *$x' = 0, x'_+ = 0, x'_- = 0$ are added to all initializing incoming transitions,*
   (c) *$x'_- = x_- - m$ is added to all outgoing transitions, and*
   (d) *$x' = x - m$ is added to all successful outgoing transitions supplementarily.*

In Fig. 4, coordination is achieved by the synchronization variable *ball*. It has capacity 1, because obviously there is only one ball in a soccer game, and is introduced in the concurrent state *teamplay*, i.e. $\gamma(\mathit{ball}) = \mathit{teamplay}$. 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. Concerning the *gotoBall* state, the transition annotated with $||\mathbf{p} - \mathbf{b}|| \leq ||\mathbf{pT} - \mathbf{b}||$ is an incoming transition. The transition marked with $||\mathbf{p} - \mathbf{b}|| \leq \mathit{DHB}$ is successfully outgoing, while the transition marked with a crossed box is failed. Since the state *gotoBall* directly belongs to the region *player*, there are no other (indirect) incoming or outgoing transitions.

3.3 Operation of Hybrid State Machines

The state machine starts with the *initial configuration*, that is the completed topmost initial state of the overall state machine. In addition, an initial condition must be given, that is a predicate with free variables from $X \cup \dot{X}$. The current *situation* of the multiagent system can be characterized by a pair (c, v) where c is a configuration and v is a valuation, i.e. a mapping $v : X \cup \dot{X} \to \mathbb{R}$. The *initial situation* at time $t = 0$ is a situation (c, v) where c is the initial configuration and v satisfies the initial condition.

The behavior of a hybrid state machine can now be described by continuous and discrete state changes. Let (c, v) be the current situation, and $S(c)$ be the set of states occurring in the configuration tree c. As long as the conjunction of the invariants of all

$s \in S(c)$ hold, the multiagent system evolves according to the conjunction of the flow conditions associated with all states $s \in S(c)$; we call this *continuous change*. Whenever after some time τ (chosen minimally) the invariants of one or more states do not hold any longer, then (and only then) a discrete state change takes place, called micro-step:

Definition 8 (micro-step). *A micro-step from one configuration c of a state machine to a configuration c' by means of a transition sTs' with some jump condition in the current situation (written $c \to c'$) is possible iff:*

1. *c contains a node labeled with s whose invariant does not hold any longer,*
2. *the jump condition of the given transition holds in the actual situation (c, v),*
3. *c' is identical with c except that s together with its subtree in c is replaced by the completion of s', and*
4. *the variables in X' are set according to the jump condition.*

We assume, that hybrid state machines are deterministic automata, i.e., for each state s, the jump conditions of all transitions outgoing from s are pairwise inconsistent with each other. Nevertheless it might happen, that after some time τ several invariants begin not to hold at the same time, then several micro-steps are performed in parallel for all respective states (called *macro-step* then). Conflicts may arise, if invariants of states on one and the same path in the configuration tree are involved. In this case, the conflict must be solved. Here, outer transitions may be preferred over inner ones. The advantage of this procedure is that the agents are more reactive. In UML statecharts inner transitions have priority over outer transitions, while this is the other way round in [3]. State transitions are triggered by the invariants.

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.

4.1 Examples with Standard Model Checkers

For the behavior specification shown in Figs. 1 and 2 we conducted several experiments with the standard model checkers HYTECH [5] and PHAVer [2]. Both model checkers are implemented for the analysis of linear hybrid automata. They take textual representations of hybrid automata like the one in Fig. 5 as input and perform reachability tests on the state space of the resulting product automaton. This is usually done by first

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

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 (w.r.t. time units, *not* discrete transitions) 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 $wLevel \leq 0$, because the state *extinguish* is only left when the water level is zero, thus including a check for equality leads to false results.

Figure 5 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 in 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 we have $civ \geq 0$.

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.

Concerning the robotic soccer example (Fig. 4), there are several questions, which can be answered with or without model checking. First of all, it is clear that because of the synchronization variable *ball* at most one agent will go to the ball. This can be seen by a careful inspection of the specification. However, the question whether always at least one agent goes to the ball, cannot be answered that easily. Therefore this is worthwhile to be model checked. This is ongoing work.

4.2 Effective Transformation of Multiagent Specifications

The original hybrid automata allow neither hierarchies nor concurrency. Hence, in order to be able to use standard hybrid model checkers, hierarchical hybrid automata as stated in this paper have to be flattened. For this, as states of the simple (flat) hybrid automaton we take the configurations c with invariants and flow conditions taken as the conjunction of the respective conditions in the states in $S(c)$. Thus, we define $flow(c) = \bigwedge_{s \in S(c)} flow(s)$ and $invariant(c) = \bigwedge_{s \in S(c)} invariant(s)$, respectively, for each configuration c. The transitions between configurations of the flat automaton can be defined as follows: there is a transition between c and c' iff a micro- or macro-step is possible. This means, there exist one or more transitions $s_1 T s'_1, \dots, s_m T s'_m$ for $m \geq 1$ in the original automaton, annotated with the jump conditions $jump_1, \dots, jump_m$, respectively, such that $c \longrightarrow c'$. Then, we simply annotate the transition from c to c' in the flat automaton with the conjunction $jump_1 \wedge \cdots \wedge jump_m$.

A problem during the transformation process is that some of the constraints, e.g. invariants, lead to heavily non-linear (in)equations, e.g. $\|\mathbf{p} - \mathbf{b}\| \geq DHB$. This cannot be dealt with standard model checkers for at least two reasons: they can neither deal practically nor even theoretically with them because of the appalling computational complexity. Therefore, the above-stated condition has to be reformulated. The Euclidean distance can be approximated by the Manhattan distance: $|p.x - b.x| + |p.y - b.y| \geq DHB$.

It should be remarked that synchronization points help us to reduce complexity. In order to see this, let us consider a multiple 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 in principle k^m different configurations, i.e. exponentially many. 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 $k = 2$.

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. 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.

References

1. Toshiaki Arai and Frieder Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In *Proceedings of 1st International Joint Conference on Autonomous Agents & Multi-Agent Systems*, pages 11–18. ACM Press, 2002.
2. Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, Proceedings*, LNCS 3414, pages 258–273. Springer, 2005.
3. David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
4. Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.
5. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: The Next Generation. In *IEEE Real-Time Systems Symposium*, pages 56–65, 1995.
6. Jan Murray. Specifying agent behaviors with UML statecharts and StatEdit. In Daniel Polani et al., editors, *RoboCup 2003: Robot Soccer World Cup VII*, LNAI 3020, pages 145–156. Springer, 2004.
7. Jan Murray and Frieder Stolzenburg. Hybrid state machines with timed synchronization for multi-robot system specification. In Carlos Bento et al., editors, *Proceedings of 12th Portuguese Conference on Artificial Intelligence*, pages 236–241. IEEE Inc, 2005.
8. Jan Murray, Frieder Stolzenburg, and Toshiaki Arai. Hybrid state machines with timed synchronization for multi-robot system specification. *KI*, 3/06:45–50, 2006.
9. Object Management Group, Inc. *UML Specification, Version 1.5*, March 2003.
10. Object Management Group, Inc. *UML 2.0 Superstructure Specification*, October 2004.
11. 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. Springer, 1991.
12. Frieder Stolzenburg and Toshiaki Arai. From the specification of multiagent systems by statecharts to their formal analysis by model checking: Towards safety-critical applications. In Michael Schillo et al., editors, *Proceedings of 1st German Conference on Multiagent System Technologies*, LNAI 2831, pages 131–143. Springer, 2003.
13. Satoshi Tadokoro 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)*, pages 4089–4104, 2000.