

UNIVERSITÄT
KOBLENZ · LANDAU



**From the Specification of Multiagent
Systems by Statecharts to their
Formal Analysis by Model Checking**

Frieder Stolzenburg

6/2001



Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz

E-mail: researchreports@infko.uni-koblenz.de,

WWW: <http://www.uni-koblenz.de/fb4/>

From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking

Frieder Stolzenburg
Universität Koblenz-Landau
Rheinau 1, 56075 Koblenz
GERMANY
stolzen@uni-koblenz.de

Abstract

A formalism for the specification of multiagent systems should be expressive and illustrative enough to model not only the behavior of one single agent, but also the collaboration among several agents and the influences caused by external events from the environment. For this, *state machines* [25] seem to provide an adequate means. Furthermore, it should be easily possible to obtain an *implementation for each agent* automatically from this specification. Last but not least, it is desirable to be able to check whether the multiagent system satisfies some interesting properties. Therefore, the formalism should also allow for the verification or formal analysis of multiagent systems, e.g. by *model checking* [6].

In this paper, a framework is introduced, which allows us to express declarative aspects of multiagent systems by means of (classical) propositional logic and procedural aspects of these systems by means of state machines (statecharts). Nowadays statecharts are a well accepted means to specify dynamic behavior of software systems. They are a part of the Unified Modeling Language (UML). We describe in a rigorously formal manner, how the specification of spatial knowledge and robot interaction and its verification by model checking can be done, integrating different methods from the field of artificial intelligence such as qualitative (spatial) reasoning and the situation calculus. As example application domain, we will consider robotic soccer, see also [24, 31], which present predecessor work towards a formal logic-based approach for agents engineering.

Key words. knowledge-based systems; multiagent systems; finite state machines; qualitative reasoning; model checking.

Introduction

In multiagent systems, often logic-based approaches based on the situation calculus are considered. There, knowledge—in particular, spatial knowledge—is represented propositionally by sets of logical formulae. This procedure yields a qualitative representation of knowledge, where also reasoning about this knowledge is possible. In this context, mainly non-classical logics—first and foremost, temporal logics—are applied. Reasoning can be performed in these logics directly, or the specification can be transformed into another appropriate format. However, in such a purely logic-based approach, it is often difficult to specify explicitly concurrent actions or the collaboration among several agents in the environment.

In this paper, we discuss the details from the specification of multiagent systems—regarding also concurrency and collaboration among agents—to their formal analysis. For this, we introduce state machines (Section 1) and provide a rigorously formal semantics for them (Section 2). Next, we show how they can be used for the specification of multiagent systems (Section 3), giving examples from the robotic soccer domain. State machines can be transformed into a logical structure, after a qualitative approximation, i.e. abstraction of the input sensor data and of the effects of actions has been done (Section 4). This allows us to apply formal verification techniques (model checking) to multiagent systems.

In summary, we combine different methods from the field of artificial intelligence, namely knowledge representation, qualitative (spatial) reasoning, cognitive robotics (situation calculus), and validation and verification of multiagent systems. In addition, by the semantics stated here, state machines can be almost directly turned into executable specifications of multiagent systems. The whole software engineering process of multiagent systems can be completely formalized. They can be specified in a hierarchically structured, modular way. As example in this paper, we will consider the *robotic soccer domain*, see also [24, 31], which present predecessor work towards a formal logic-based approach for agents engineering.

1 State Machines

State machines are ubiquitous in computer science and robotics (see e.g. [4, 15, 25]), because they provide a way to define a mapping between the internal state of an agent and its operation in the world. They give us a set of concepts that can be used for modeling behavior through finite state-transition systems. They can be used to model the behavior of individual entities as well as to define the interaction or cooperation among them, which is required for adequate modeling of multiagent systems. State machines are a good way to encode procedural knowledge of how to achieve some goal, especially when conditioned actions are included.

1.1 States

Let us now formally define the basic components of state machines. For the ease of presentation, we restrict our attention to the essential ingredients of state machines. For instance, we will not consider (static or dynamic) *pseudo-states* (and final states) here explicitly, although they are present in UML [25]. The reason for this is that, in principle, pseudo-states are only used in order to abbreviate notation, allowing us to specify complex decision trees more elegantly. Alternatively, pseudo-states may be viewed as simple transitory states.

The basic components of a *state machine* are the following four finite and pairwise disjoint sets:

S : the 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 : the set of events,

V : the set of variables, where each variable $v \in V$ is associated with a domain $dom(v)$, that must be either a finite domain, i.e. a subset of the integers \mathbb{Z} , or the set of real numbers \mathbb{R} —therefore, V can be naturally partitioned into two sets $V_{\mathbb{Z}} = \{v \mid dom(v) \subseteq \mathbb{Z}\}$ and $V_{\mathbb{R}} = \{v \mid dom(v) = \mathbb{R}\}$ —, and

A : the set of actions, which sometimes may be composed from a sequence of simpler actions.

States are structured hierarchically. Following the lines of [13, 25], 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 $s \in S \setminus \{s_0\}$ 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. We disallow recursive state machines by establishing the constraint that there is no state s such that $s = \beta(\dots\beta(s)\dots)$ for any number of applications of β . Note that we do not consider *final states* here. They are technically not needed, because, so to speak, agents shall act forever in their environment without stopping.

Let us now introduce the graphical notation for statecharts (as in UML): A state is drawn as rectangle with rounded corners. The name is written in the middle of the rectangle for simple states, in a rectangle resting on the outside of the top side of the state for composite states, or at its upper-left corner for concurrent states. A composite state s consists of all states s' which belong to s (i.e. $s = \beta(s')$); they are drawn within the state s . Therefore, for each $s \in S_{composite} \cup S_{concurrent}$, we define $\gamma(s) = \{s' \mid \beta(s') = s\}$; it must hold either $\gamma(s) \subseteq S_{simple} \cup S_{composite}$ or $\gamma(s) \subseteq S_{concurrent}$. In the latter case, if a composite state s consists solely of concurrent states s' , then for each such state a *region* in s has to be provided, which are separated by dashed lines.

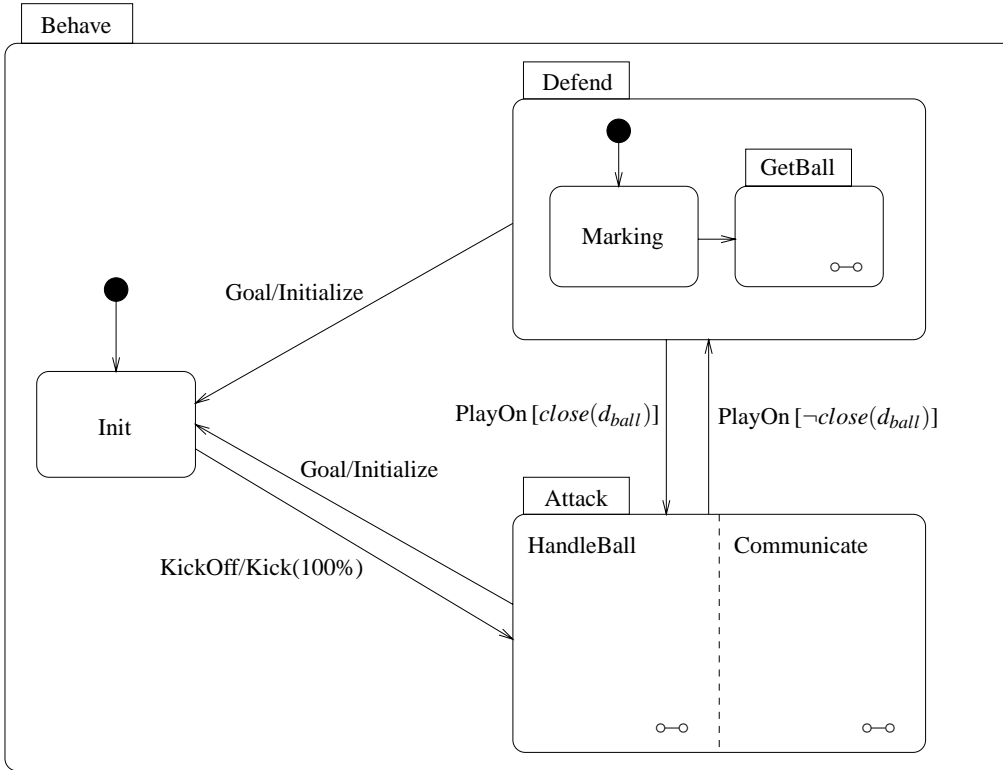


Figure 1: State machine for the overall behavior of soccer agents.

Example 1.1

The state machine in Figure 1 sketches the overall behavior of our example soccer agents. Some details are not shown, which is indicated by the *hidden decomposition icon* $\circ-\circ$. 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 s_0 is `Behave`, whose initial state is $\alpha(\text{Behave}) = \text{Init}$. Further, it holds $\alpha(\text{Defend}) = \text{Marking}$. This is marked by an edge coming from a pseudo-state (drawn as bullet \bullet) without any incoming edges. The states `Marking` and `GetBall` belong to `Defend`, i.e. $\beta(\text{Marking}) = \beta(\text{GetBall}) = \text{Defend}$. In addition, it holds $\beta(\text{HandleBall}) = \text{Attack}$ and $\beta(\text{Communicate}) = \text{Attack}$. Finally, we have $\gamma(\text{Behave}) = \{\text{Init}, \text{Attack}, \text{Defend}\}$, $\gamma(\text{Attack}) = \{\text{HandleBall}, \text{Communicate}\}$ and $\gamma(\text{Defend}) = \{\text{Marking}, \text{GetBall}\}$.

1.2 Transitions

States are connected via conditioned transitions. 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 certified conditions—called *guards*—are satisfied. 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 as defined in Definition 1.3), and $a \in A$.

At first, let us consider the guards in more detail. Actually, UML provides a means for specifying conditions with the Object Constraint Language (OCL). However, here we define guards simply as first-order formulae with the connectives \wedge , \vee , \neg , \exists and \forall , which have a standard logical semantics. In the following we use standard notions and notations of first-order logic [11, 18].

Definition 1.2 (Term)

The sets of integer terms and real-valued terms, respectively, are defined as follows. We must distinguish different variable and term sets.

1. The set of *integer terms* $T_{\mathbb{Z}}$ is the smallest set containing $V_{\mathbb{Z}}$ and \mathbb{Z} (the set of integer constants) such that if t_1 and t_2 are integer terms, then also $t_1 + t_2$ and $t_1 \cdot t_2$ are integer terms.
2. The set of *real-valued terms* $T_{\mathbb{R}}$ is the smallest set containing $V_{\mathbb{R}}$ and \mathbb{Z} such that if t_1 and t_2 are real-valued terms, then also $t_1 + t_2$ and $t_1 \cdot t_2$ are real-valued terms.

The set of terms T can be defined simply as the union of $T_{\mathbb{Z}} \cup T_{\mathbb{R}}$.

Definition 1.3 (Guard)

The set of guards G is the smallest set such that:

1. For all real-valued terms $t_1, t_2 \in T_{\mathbb{R}}$, $t_1 = t_2$ is in G .
2. For all integer terms $t_1, t_2 \in T_{\mathbb{Z}}$, $t_1 \doteq t_2$ is in G .
3. If g_0, g_1 and g_2 are in G , then also $\neg g_0$, $g_1 \wedge g_2$ and $g_1 \vee g_2$ are in G .
4. For $g \in G$ and $v \in V$, we have that $\exists v g$, and $\forall v g$ are in G .

The semantics of guard formulae (and terms) can now be defined easily by the standard semantics of first-order formulae. For this, the reader is referred to standard textbooks [11, 18]. Clearly, the overall domain in this context is $\mathbb{Z} \cup \mathbb{R}$; the interpretation function I must interpret variables according to their special domains, i.e. $I(v) \in \text{dom}(v)$; the symbols \doteq and $=$ denote equality in \mathbb{Z} and \mathbb{R} , respectively; the symbols $+$ and \cdot are overloaded and denote addition and multiplication, respectively, in \mathbb{Z} or \mathbb{R} . Since we aim at a finite representation, equations of the first type will be approximated later on by equations of the second type (see Section 4.1). In order to improve readability, we will also allow shorthands for equations as in Example 1.6 (see below).

The following theorem shows that validity of guard formulae is decidable. Hence, in principle, for the implementation of (single) agents and for the analysis of multi-agent systems we could employ constraint solvers for testing guard formulae. However, the theoretical complexity of solving first-order problems with real-valued variables is

quite high (see [29]). This gives us another motivation for approximating real-valued variables by variables over finite domains, because this reduces the computational complexity.

Theorem 1.4

The validity of a guard formula is decidable.

Proof.

We give only a sketch of the proof here. Let us first eliminate all occurrences of integer variables. For this, we replace in g , for all $v \in V_{\mathbb{Z}}$, all formulae $\exists v g_1$ and $\forall v g_2$ by $\bigvee_{z \in \text{dom}(v)} g_1[v/z]$ and $\bigwedge_{z \in \text{dom}(v)} g_2[v/z]$, respectively, where $g_i[v/z]$ denotes the formula g_i after replacing all (free) occurrences of v by z . This procedure yields us a formula g' . It is easy to see, that g and g' are equivalent, i.e. $I(g) \equiv I(g')$. Since g' contains only variables $v \in V_{\mathbb{R}}$, the problem is reduced to the validity problem of the first-order theory of reals. But this problem is decidable, which can be inferred from Tarski's famous theorem [32] that the theory $(\mathbb{R}, +, \cdot)$ is decidable. \square

Definition 1.5 (Transitions)

Let N be the set of all annotations. The set of transitions of a state machine is a ternary relation $T \subseteq S \times N \times S$, such that if $(s, n, s') \in T$, then s and s' belong to the same state, i.e. $\beta(s) = \beta(s')$, and neither s nor s' are concurrent states. In addition, for each simple or composite state s (except for the start state s_0), there is at least one transition starting or ending in s .

1.3 Robotic Soccer as Example Domain

As example domain, we choose (robotic) soccer, since it is illustrative and it finds also practical application in the *RoboCup*, which is an international research and education initiative, attempting to foster artificial intelligence and intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined. The author of this paper is involved in the simulation league (see e.g. [24, 31]) with simulated soccer agents using the *Soccer Server* [8].

Example 1.6 (Soccer domain)

In the following, we consider the following basic elements of a multiagent system for robotic soccer. We distinguish the play modes (i.e. events in E) KickOff, Goal (when somebody shot a goal) and PlayOn. We consider the (parameterized) actions Initialize (moving to the center point), Turn(Angle in $^\circ$), Kick(Power in %), and Dash(Power in %) (accelerating agent) in A . The set V contains the following variables v :

v	$dom(v)$	meaning
m, n	$\{1, \dots, 11\}$	uniform number of teammate
p	$\{0, \dots, 100\}$	kick or dash power
a_{ball}, a_{goal}	$\{-180, \dots, +180\}$	angle to ball and opponent goal
a_1, \dots, a_{11}	$\{-180, \dots, +180\}$	angle to teammate with respective number
d_{ball}, d_{goal}	\mathbb{R}	distance to ball and opponent goal
d_1, \dots, d_{11}	\mathbb{R}	distance to teammate with respective number

For the ease of notation, we introduce some shorthands: If l and r are real-valued terms, then $l \leq r$ is a shorthand for $\exists v (l + v \cdot v = r)$ where v is a (new) variable in $V_{\mathbb{R}}$. Furthermore, $r \geq l$ means $l \leq r$, and $l < r$ means $\neg(l \geq r)$. With these abbreviations, we can define the following predicates, where x denotes a real-valued term:

shorthand	formula	meaning
$close(x)$	$x \leq 1$	within kicking distance for agent
$distant(x)$	$x \geq 30$	pass beyond this distance not possible

Let us consider Figure 1 again. The three states Init, Attack and Defend belong to the main (start) state Behave. Its initial state $\alpha(\text{Behave}) = \text{Init}$ permits a transition annotated with KickOff/Kick(100%) (with empty guard which corresponds to *true*) to Attack, if the agent has a KickOff from the center point. In this case, the agent kicks the ball with full power and enters the Attack state afterwards. Attack and Defend states are mutually connected by transitions without actions: if the agent is in ball possession, then it is in the Attack state; otherwise it is in Defend state. Further, there are transitions from Attack and Defend back to Init, if somebody scores a goal.

2 How do State Machines Work?

In this section, we give an overview on how state machines work. Since 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, we will first give a rather abstract view of working (systems of) state machines: they change the situation they are in, forming a trace, which is a sequence of situations (see Section 2.1).

After this, we consider state machines and their effects more concretely. State machines have an internal state, called configuration (defined below). They perform steps from one configuration to another (see Section 2.2). We will distinguish between micro- and macro-steps as in [27]. In addition, for proper multiagent systems, we also must take into account that several agents may perform steps at the same time.

Definition 2.1 (Configuration)

A *configuration* c is a tree of states rooted at a node (labeled with) s_0 , such that if a node s' is an immediate ancestor of node s , then state s belongs to s' , i.e. $s' = \beta(s)$. A configuration c is called *complete*—written \underline{c} —iff for every node labeled with s :

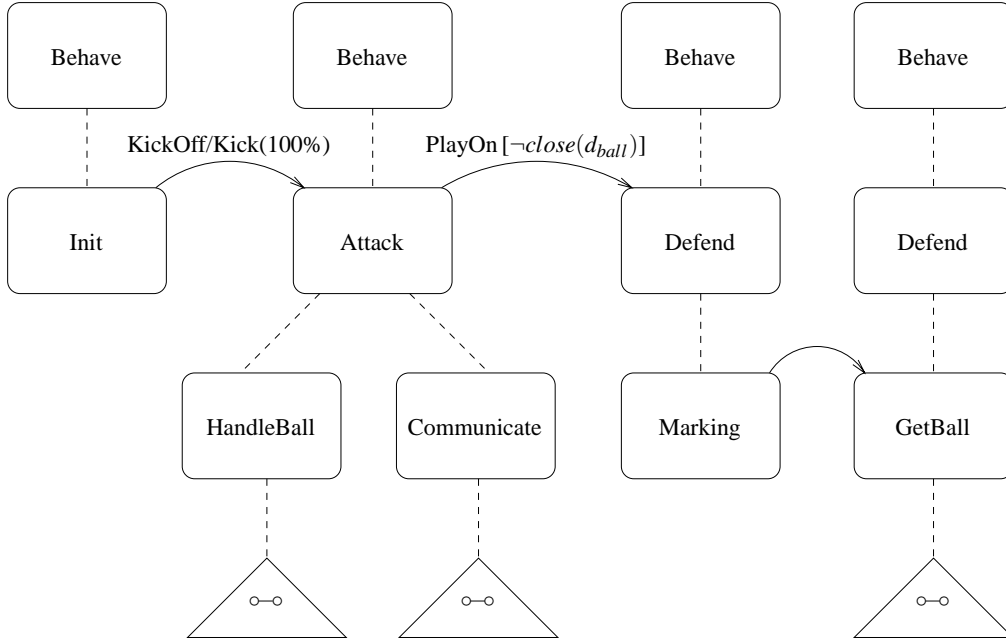


Figure 2: Configurations as trees.

1. If s is a composite state, then the configuration c must contain $\alpha(s)$ as immediate successor of s .
2. If s is a concurrent state, then for each s' such that $s = \beta(s')$, c must contain s' as immediate successor of s .

Example 2.2

In Figure 2, four configurations for the state machine in Figure 1 are shown, where nodes are connected via dashed lines. The leftmost configuration is the initial one. The root of this configuration tree is clearly labeled with Behave. But since Behave is a composite state, the configuration has to be completed according to Definition 2.1, by adding the initial state Init as immediate successor of Behave. Since Init is a simple state, the configuration tree is complete afterwards. We will continue this example later on (see Example 2.4).

2.1 Situations

A *situation* is given by an assignment $\sigma : V \rightarrow \mathbb{Z} \cup \mathbb{R}$, mapping each variable $v \in V$ into an element of its corresponding domain $dom(v)$. Situations change over time as an effect of actions the agents are performing. With Σ , we denote the set of all situations. We write $\sigma \models (l = r)$ iff $\sigma(l) = \sigma(r)$. This means, evaluating the left and the right sides of equations, where variables are evaluated according to σ , yields us the same

result. This definition can be extended to guard formulae in general in a straightforward manner.

In order to be able to formalize systems of multiple agents, say k agents, we require the set of variables V to contain at least variables (with finite domains) for the actual event—we assume that only one event occurs at a time—and the (internal) configuration of each agent. One agent can perform actions in different concurrent states (see Section 2.2), and several agents can act in parallel (see Section 2.3). Hence we introduce an action variable for each agent, storing the actions done by the respective agent during the step. The values of these variables are updated after each step; this means, we adopt a discrete time model here. The following table summarizes, what we have just said.

v	$dom(v)$	meaning
$event$	E	current event
$conf_1, \dots, conf_k$	C	configuration of each agent
$action_1, \dots, action_k$	2^A	actions of each agent

2.2 Micro- and Macro-Steps

Before we come back to the formalization of the behavior of multiagent systems in general, let us concentrate on the behavior of a single agent for a while. The activity of a single agent can be described by a sequence of configurations, where in each micro-step exactly one transition is applied. This means, that we interpret concurrent actions only as quasi-parallel, by sequentializing them. It follows a definition of a micro-step.

Definition 2.3 (Micro-step)

A *micro-step* from one configuration c of a state machine to another configuration c' by means of some transitions $t = (s, n, s')$ with $n = e[g]/a$ in a situation σ —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 s' , i.e. the completion of s' , and
3. $\sigma \models event \doteq e \wedge g$ holds.

Example 2.4

Let us revisit Figure 2 again. 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 Figure 2 is possible according to Definition 2.3. 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 $\circ\text{--}\circ$ in it. Two other configurations and steps are shown in Figure 2; the reader may explore them on his or her own.

State machines should be locally conflict-free, i.e., for all configurations c and steps $c \rightarrow_{t_1} c_1$ and $c \rightarrow_{t_2} c_2$ where $t_1 = (s, e_1[g_1]/a_1, s_1)$ and $t_2 = (s, e_2[g_2]/a_2, s_2)$, we have that $e_1 \neq e_2$, or g_1 and g_2 are logically inconsistent. Note that this constraint neither implies that the behavior of an agent is deterministic nor that each agent can perform only one action at a time (see Definition 2.5, where transitions from *different* states are allowed). A similar (but weaker) notion is taken into consideration in [13]. There, *priorities* are introduced in order to resolve the remaining conflicts. But in order to keep things technically as simple as possible, we will not consider explicit procedures for handling this type of priorities here. They could be simulated in our framework by using special variables in V , which stand for the priority of transitions.

Since we speak of steps of state machines, we have a discrete model of time for multiagent systems. In addition, we adopt the so-called *synchrony hypothesis*, 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. In general, one step may contain several micro-steps for each agent, called macro-step, which we define next.

Definition 2.5 (Macro-step)

Let c be a configuration and micro-steps with the transitions $(s_1, n_1, s'_1), \dots, (s_l, n_l, s'_l)$ be given. Then, a *macro-step* from the configuration c to a configuration c' with the given micro-steps is possible iff:

1. all micro-steps are possible in c according to Definition 2.3,
2. all of them are simultaneously applicable, i.e., all states s_1, \dots, s_l occur in different paths in the configuration tree c , and
3. c' is obtained by applying all given micro-steps to c .

In addition, we adopt the assumption that there is no micro-step possible with a transition (s_*, n_*, s'_*) , such that s_* is a label of a node which is a not necessarily immediate ancestor of one of the s_i with $1 \leq i \leq l$. This means, we have an implicit priority mechanism along the hierarchy of composite states. In UML statecharts inner transitions of a state machine have priority over outer transitions, while this is the other way round in [13]. Thus, we take the latter view here and in the implementation of our RoboLog system (see Section 3.4), because then our agents can react more instantly to exogenous events and interrupts. In [20], priorities are not fixed, but they can be specified by the user.

Since we aim at a finite representation of multiagent systems, we must make sure that there are only finitely many configurations, and all of them are finite. But this gives us the following theorem.

Theorem 2.6

All configurations of a state machine are finite, and there are only finitely many different configurations of a state machine.

Proof.

Let us first show the first part of the proof, that all configurations are finite. First, we observe that for each node in a configuration (except the root) labeled with state s , it must hold that its immediate ancestor in the configuration tree is labeled with $\beta(s)$. This follows from the definition of complete configuration (Definition 2.1) and the fact that, for any transition (s, n, s') , s and s' must belong to the same state (according to Definition 1.5), because s can be replaced by s' in a step of the state machine (Definitions 2.3 and 2.5).

Now, since recursive state machines are disallowed, it is clear that along a path in a configuration tree, all nodes in it must be labeled with different states. Finally, since there are only finitely many states in S , each path must be finite, and therefore the configuration tree itself must be finite. Note that there are always only finitely many concurrent regions of a composite state.

The second part of the theorem, that there are only finitely many different configurations, is quite easy to show. This follows simply, because we can only build finitely many trees—which are always finite as we have just shown—from finitely many states. This completes the proof. \square

2.3 Traces

Now, we are ready to formalize the behavior of a whole system of multiple agents. In general, each agent may have its own state machine with separate local variables. Therefore, for each agent i with $1 \leq i \leq k$, we have an associated set of variables. In principle, each agent has only access to its own set of variables V_i —these are among others the indexed variables in our context, e.g. $conf_i$ and $action_i$ —and some global variables V_0 , e.g. $event$. The set of all variables V is now the disjoint union of V_0 and V_1, \dots, V_k .

In order to describe the behavior of the whole system, we must consider the values of all variables in V of course. As just said, in general, each agent may have its own state machine with start state s_1, \dots, s_k . Therefore, the *start situation* σ_0 must satisfy the following condition:

$$\sigma_0 \models conf_1 \doteq \underline{s_1} \wedge \dots \wedge conf_k \doteq \underline{s_k}$$

Next, we will introduce the notion of trace for a multiagent system (Definition 2.8). Situations change because agents perform actions whose effect is to change the values of variables in V , in particular the configurations of the agents. This is formalized in the subsequent definition. Note that the sequence of situations is not completely determined by the conditions in Definition 2.7. They just give necessary conditions for the change relation \rightsquigarrow defined next.

Definition 2.7 (Change relation)

The relation \rightsquigarrow between situations, called *change relation*, must satisfy the following conditions:

1. If $\sigma \models \text{conf}_1 \doteq c_1 \wedge \dots \wedge \text{conf}_k \doteq c_k$, then $\sigma' \models \text{conf}_1 \doteq c'_1 \wedge \dots \wedge \text{conf}_k \doteq c'_k$, where, for each i with $1 \leq i \leq k$, either $c'_i = c_i$ or c'_i is obtained by a macro-step from c_i wrt. the state machine of agent i .
2. If $\sigma \models \text{action}_1 = a_1 \wedge \dots \wedge \text{action}_k = a_k$, then $\sigma' \models \text{effect}(a_1) \wedge \dots \wedge \text{effect}(a_k)$, where $\text{effect}(a)$ denotes the predicate for the set of actions a , stating the effect of performing all actions in a .

The last conditions can be related to the situation calculus (see also Section 5.1). However, there in order to do planning it is assumed that the effect of an action is completely known and thus deterministic. We will not make this assumption here. In practice, this uncertainty is solved by the sensors of the agent giving the agent always the actual data. In theory, the non-determinism is just reflected by a non-determinism of transitions in state machines and later on in Kripke structures for model checking (see Section 4.2).

Definition 2.8 (Trace)

A *trace* of a multiagent system is a (possibly infinite) sequence of situations $\sigma_0, \sigma_1, \dots$ in Σ , where σ_0 is the start situation and $\sigma_n \rightsquigarrow \sigma_{n+1}$ for all $n \geq 0$.

3 From Single Agents to Multiagent Systems

In the sequel, we will show how important aspects of multiagent systems can be implemented by variables, while showing their graphical representation in a state machine diagram. We will start with examples for single agents, later on also coming to examples with multiagent collaboration. Many concepts, such as the synchronization of several regions of an agent, cooperation among agents, and interrupts of behaviors can be treated in the same way. In UML, synch states and the history mechanism are introduced for denoting synchronization and interrupts. There are also special kinds of diagrams for the specification of interaction (cooperation) among several agents (see e.g. [25, 26]). Nevertheless, we prefer to show all aspects of an agent within one (type of) diagram.

In essence, all that these approaches only provide are some notation whose semantics can be mimicked by what we already have, namely (special) variables in V and actions changing them. For this purpose, we require that V contains the variables *event* and *action* denoting the actual event and action plus special variables for synchronization and other purposes, which we introduce in the following examples. They highlight the topics of synchronization (Section 3.1), interrupts (Section 3.2) and multiagent collaboration (Section 3.3).

3.1 Passing the Ball with Synchronization

Let us assume that our soccer playing agents have modules for moving and kicking that can be controlled independently from each other. Then, in order to perform a pass,

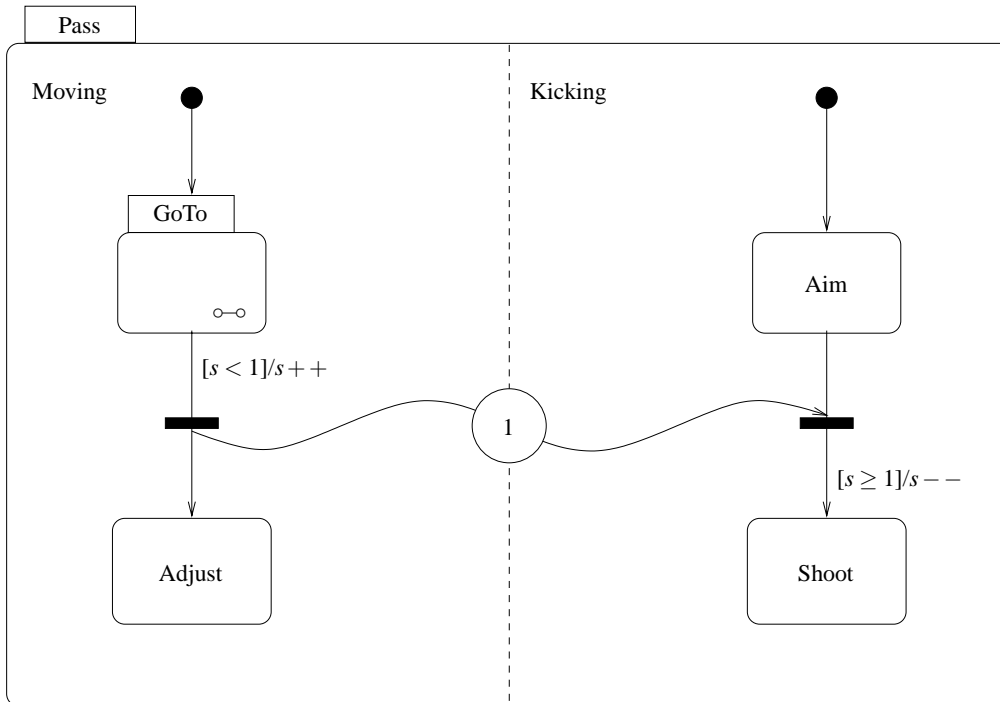


Figure 3: Machine with concurrent states for Passing.

an agent may first go to the ball, while it aims at the opponent goal with its kicking device all the time. If the ball is reached, the shot can be initiated and also the position of the agent can be re-adjusted. Observe that the last action requires synchronization between the Moving module and the Kicking module. This is shown in Figure 3.

For the synchronization of (two) concurrent regions of a state machine, synch states are introduced in UML. A *synch state*, which is drawn as a circle, is always used in conjunction with fork and join states, which are shown as thick bars **█**. In Figure 3, the left bar is a *fork state*, while the right one is a *join state*. The firing of outgoing transitions from a synch state can be limited by a specified bound $b > 0$ on the difference between the number of times outgoing and incoming transitions have fired. In the example, this bound is set to 1, because there is only one ball that can be kicked.

We can simulate synchronization by introducing a new variable s with $dom(s) = \{0, \dots, b\}$ for each synch state, which must be 0 initially. The effect of the transition via the synch state can be expressed by setting up the (additional) conditions and actions along the transitions that go through the fork and join states as shown in Figure 3, where $s++$ and $s--$ mean incrementing and decrementing, respectively, the value of s . The introduction of s makes the use of the synch state superfluous. Note that only these additional annotations for the treatment of synchronization are shown. All other transitions and annotations are not shown here for the ease of representation.

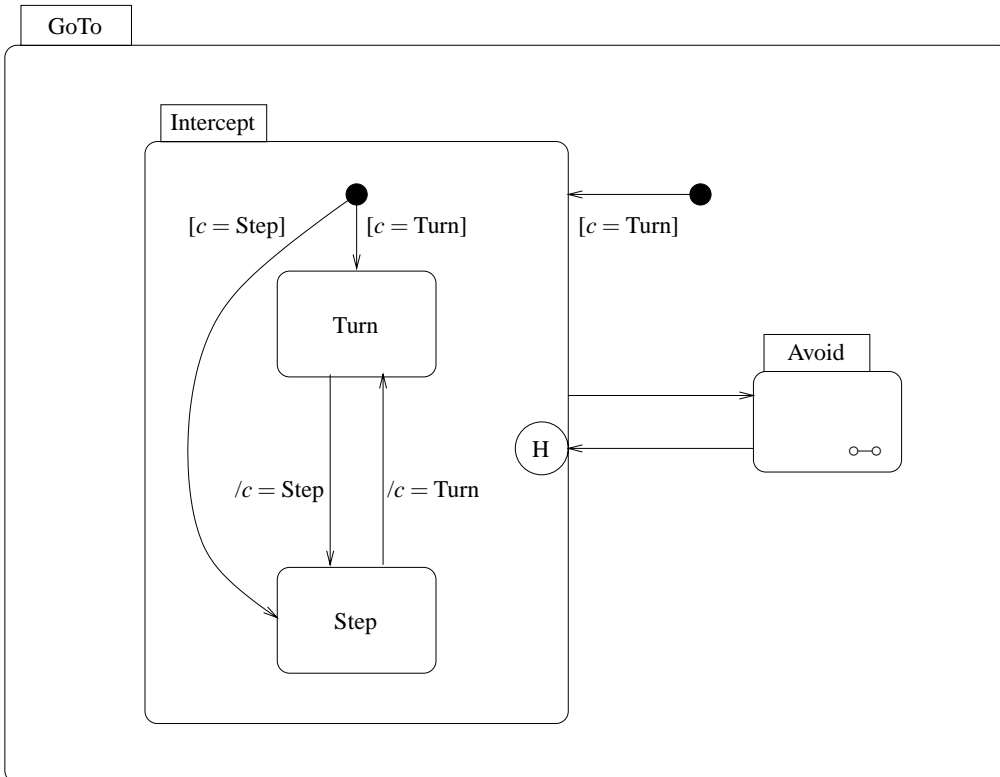


Figure 4: Obstacle avoidance using history states.

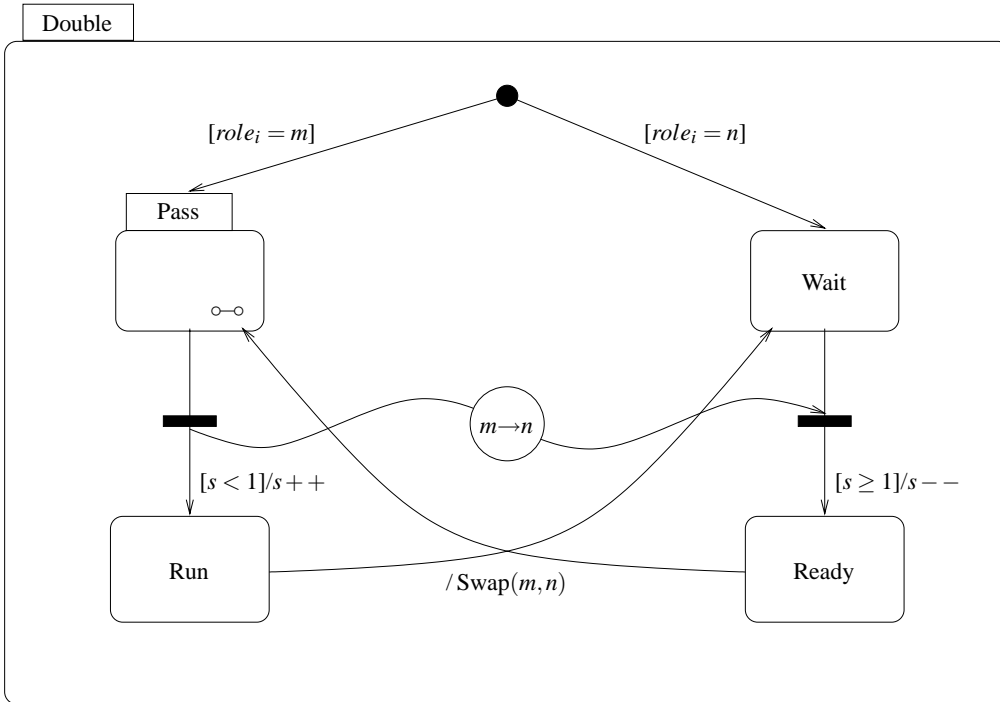


Figure 5: Collaborating agent script for double passing.

3.2 Obstacle Avoidance with Interrupts

Let us now consider the state `GoTo` in more detail. 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 Figure 4. 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 c for each history state where $dom(c)$ is the set of all configurations of s . Note that $dom(c)$ is finite because of Theorem 2.6. For each transition in state s , the value of c has to be updated accordingly. In addition, when re-entering s , the state processing has to be continued with the stored configuration. Look at Figure 4. Again, only the additional annotations for the treatment of the interrupt are shown in the figure. Note that if s is a nested state with concurrent states belonging to it, then the update procedure has to be refined accordingly.

3.3 Collaboration for Double Passing

The interaction of agents can be considered analogously to the synchronization of modules within one agent. As example, let us consider a double passing, as shown in Figure 5. There are two agents m and n involved in the situation. The first one passes the ball to the other one, provided that the other agent is already waiting for it. The first agent runs towards the opponent goal, while the other agent is ready for passing the ball back to him. This means that both agents continuously change their roles, which is indicated by the Swap action.

Recall that each agent runs its own state machine. We assume in this example, that both agents run a copy of the same state machine with start state Double. Nevertheless, each agent has its own variables. Clearly, the synchronization variable s belongs to the set of global variables V_0 , whereas $role_i$ is a local variable for both agents, it indicates the role for each agent (pass or receive).

The interesting thing is that the interaction among the agents can be shown in one statechart diagram. However, by referring to the value of the variable $role_i$, they may behave differently. Again, in order to model the whole system adequately, we have to notice, that each agent has its own set of variables, while several variables are shared.

3.4 Agent Programming

The previous examples show that modeling complex behavior for a single agent can be done conveniently by means of state machines. State machines or their relatives can also be taken more or less directly as executable specifications for agents, where the sensor inputs are given via some variables in V , and the actions in A correspond to real effects of the agent.

Since guards are logical formulae, it seems to be a good idea to implement state machines in a logic programming language such as *Prolog*. This is done in [24, 31], where an approach for developing soccer agents declaratively is presented. The advantage of making use of Prolog is that by this, guard conditions can be expressed adequately, yielding us a rule-based specification. For instance, the decision process (decision trees) can be programmed conveniently in prolog. Last but not least, agent programmers have a full programming language available and can design and implement agents without many restrictions.

In the RoboLog system [23, 24], the agent control procedure is realized in prolog, whereas the basic functionality is implemented in C++. The system contains an explicit state machine, i.e. a component for performing macro-steps employing the implicit priority that outer transitions are preferred (see also Section 2.2). A user can define multiagent systems by specifying the state hierarchy and transitions in Prolog. However, there does not yet exist a graphical user interface. More details on how statechart specifications can be transformed into running code can be found in [22].

4 Analyzing Multiagent Systems

In order to explain the behavior of an agent in its environment with possibly more than one agent, it is necessary to take into account (in a formal manner) the behavior of all agents in the environment. We already did this in Section 2.3. After having seen some examples for statecharts specifying single agents and multiagent behavior in Section 3, we will now investigate how we can formalize the whole system such that verification and system analysis by model checking is possible. For this, we introduce (in Section 4.1) a method for obtaining a completely finite system description. This can be translated in to a Kripke structure (see Section 4.2). By this, formal analysis by means of temporal logics is possible (see Section 4.3).

4.1 Qualitative Reasoning

With traces, we are able to describe the behavior of multiagent systems. However, since some of the variables have infinite domains, reasoning about system properties does not seem to be really feasible. Because of this, we approximate real-valued variables by variables with finite domains, which is called *data abstraction* [6]. Therefore, for each variable $v \in V$ we introduce (new) finite domains $\overline{dom}(v)$, where $\overline{dom}(v)$ is finite, if $dom(v) = \mathbb{R}$; otherwise, $\overline{dom}(v) = dom(v)$. In the sequel, we present two possibilities of approximating guards (in Definition 4.1 and Definition 4.4).

Definition 4.1 (Existential approximation)

For each $v \in V_{\mathbb{R}}$, let a function $h_v : \mathbb{R} \rightarrow \overline{dom}(v)$ be given. Let v_1, \dots, v_n be free and real-valued variables in a guard g . Let now $\hat{v}_1, \dots, \hat{v}_n$ be new variables in $V_{\mathbb{Z}}$ with $dom(\hat{v}_i) = \overline{dom}(v_i)$ for $1 \leq i \leq n$. Then, the formula \hat{g} defined as

$$\exists v_1 \dots \exists v_n (\hat{v}_1 \doteq h(v_1) \wedge \dots \wedge \hat{v}_n \doteq h(v_n) \wedge g')$$

is called the *approximation* of g .

Example 4.2

Let us approximate the domain of distances d by $D = \{close, middle, distant\}$ with:

$$h_d(x) = \begin{cases} close, & x \leq 1 \\ middle, & 1 < x < 30 \\ distant, & x \geq 30 \end{cases}$$

Then, the formula $g: 0 < d_{ball} \leq 1 \wedge 0 < d_m \leq 1 \wedge d_{ball} + d_m = d$ can be approximated by the following formula \hat{g} :

$$\exists d_{ball} \exists d_m \exists d \hat{d}_{ball} \doteq h_d(d_{ball}) \wedge \hat{d}_m \doteq h_d(d_m) \wedge \hat{d} \doteq h_d(d) \wedge g$$

This formula is equivalent to

$$\begin{aligned} & (d_{ball} = close \wedge d_m = close \wedge d = close) \\ \vee & (d_{ball} = close \wedge d_m = close \wedge d = middle) \end{aligned}$$

which is a formula without real-valued variables. This can always be done as the following theorem reveals.

Theorem 4.3

Every approximation \hat{g} is equivalent to a formula without occurrences of real-valued variables (and hence without occurrences of $=$).

Proof.

First, we observe that \hat{g} does not contain any free occurrences of real-valued variables. Thus all free variables in \hat{g} range over finite domains, and there are only finitely many such variables in \hat{g} , say $\hat{v}_1, \dots, \hat{v}_n$. Hence, the number of situations σ with $\sigma \models \hat{g}$ is finite, and each of these σ can be written as a finite mapping $[\hat{v}_1 \mapsto x_1, \dots, \hat{v}_n \mapsto x_n]$. Therefore, \hat{g} is obviously equivalent to \dot{g} :

$$\bigvee_{\sigma \models \hat{g}} (\hat{v}_1 \doteq x_1 \wedge \dots \wedge \hat{v}_n \doteq x_n)$$

□

The definition of \dot{g} induces a way to construct an appropriate approximation for a given formula g . However, it may often be difficult to determine all the solutions σ such that $\sigma \models \hat{g}$. Because of this, we will now give another alternative way of finding an abstraction \tilde{g} for a formula g .

Definition 4.4

As in Definition 4.1, let a function $h_v : \mathbb{R} \rightarrow \overline{dom}(v)$ be given, for each $v \in V_{\mathbb{R}}$. For all real-valued variables $v \in V_{\mathbb{R}}$, we introduce one single domain 2^D where

$$D = \bigcup_{v \in V_{\mathbb{R}}} \overline{dom}(v)$$

Now we can establish tables for the operators $+$ and \cdot , showing the possible (approximate) results of performing the operations in the new domain. We understand them as functions $\circ : 2^D \times 2^D \rightarrow 2^D$ (where $\circ \in \{+, \cdot\}$) such that:

$$X \circ Y = \{z \mid \exists u, v, w \in V \exists x, y \in \mathbb{R} h_u(x) \in X \wedge h_v(y) \in Y \wedge z = h_w(x \circ y)\}$$

Then, the approximation \bar{g} for a guard formula g is obtained by simply replacing all occurrences of $=$ by \doteq , while changing the semantics of the operators $+$ and \cdot .

Example 4.5

Let us consider the last part of the formula from Example 4.2 $d_{ball} + d_m = d$. If we assume only non-negative values for distances, we may set up the following table, which can easily be extended to 2^D .

$+$	<i>close</i>	<i>middle</i>	<i>distant</i>
<i>close</i>	{ <i>close, middle</i> }	{ <i>middle, distant</i> }	{ <i>distant</i> }
<i>middle</i>	{ <i>middle, distant</i> }	{ <i>middle, distant</i> }	{ <i>distant</i> }
<i>distant</i>	{ <i>distant</i> }	{ <i>distant</i> }	{ <i>distant</i> }

Definition 4.4 is simpler than the previous one, since only once (and for all) tables for $+$ and \cdot have to be computed, while in Definition 4.1 each guard has to be abstracted separately. Unfortunately, \hat{g} and \tilde{g} are not equivalent (see Theorem 4.6). In this context, several approaches from qualitative spatial reasoning may be employed, which provide tables for the addition of qualitative distances or orientations (see e.g. [7, 33]).

Theorem 4.6

For all (abstracted) situations σ and guard formulae g —where σ must be defined at least for all free variables in the approximations \hat{g} and \tilde{g} of g according to Definitions 4.1 and 4.4, respectively—it holds that $\sigma \models \hat{g}$ implies $\sigma \models \tilde{g}$. The converse does not hold.

Proof.

The proof can be adapted from [6, Theorem 23]. □

4.2 Formal Analysis with Kripke Structures

Finally, we arrived at a completely finite representation of the whole multiagent system. We will transform this into a Kripke structure. This eventually allows us to reason over multiagent systems, by making use of temporal logics, e.g. Computation Tree Logic (CTL). In our context, we obtain an appropriate Kripke structure by considering all traces in the approximated domains. For further information about Kripke structures and temporal logics, the reader is referred to [6, 10].

Definition 4.7 (Kripke structure)

A *Kripke structure* consists of a finite set of states S , a set S_0 of initial states, a transition relation $R \subseteq S \times S$ that must be *right total* (i.e., for every state $s \in S$ there is a state $s' \in S$ with $R(s, s')$), and a function $L : S \rightarrow 2^P$ 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 $v \doteq x$ with $v \in V$ and $x \in \text{dom}(v)$.

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 start with the finite approximation of the given multiagent system where the domains of the real-valued variables $v \in V$ are replaced by finite domains according to Definition 4.1. We can now define the following Kripke structure:

1. The set of states S is the set of all valuations for V where all domains have been approximated by finite domains as in Section 4.1.
2. The set of initial states is a subset of all valuations which can be understood as situations σ satisfying $\text{conf}_1 = \underline{s}_1 \wedge \dots \wedge \text{conf}_k = \underline{s}_k$ where s_1, \dots, s_k are the respective start states of the state machines for the agents.
3. The labeling function labels nodes with elements in Σ .

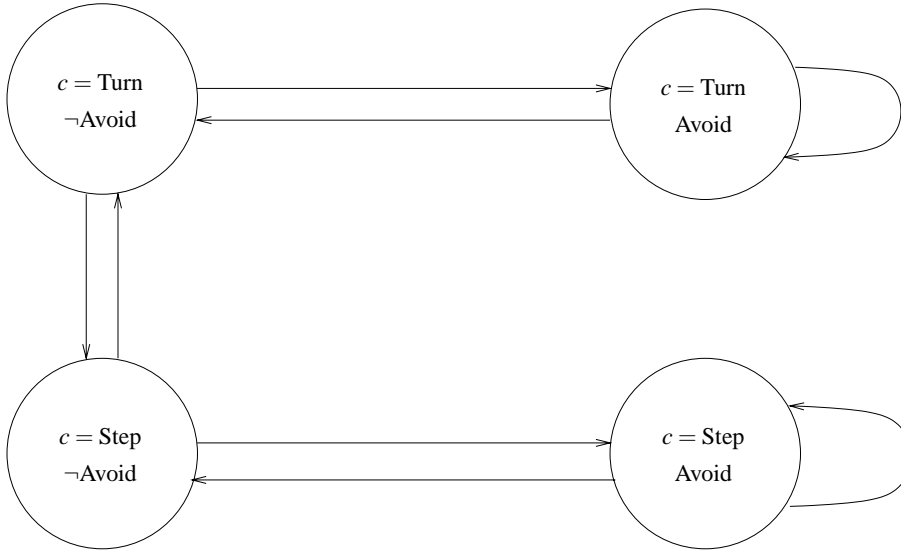


Figure 6: Kripke structure for Example 3.2.

4. Let $s, s' \in S$, then $R(s, s')$ holds, if $L(s) \rightsquigarrow L(s')$.
 If, for some s , there is no s' such that $R(s, s')$, then we add $R(s, s)$.

Example 4.8

Figure 6 shows a Kripke structure modeling some aspects of the state machine from Example 3.2. One can see, that the agent steps and turns alternately, unless it must do some obstacle avoidance. Here, the Boolean variable Avoid denotes whether the agent is in the Avoid state.

4.3 Model Checking

Now we are ready for the formal verification of multiagent systems. By means of CTL, we are able to express critical system properties. Usually, a transformation into Binary Decision Diagrams (BDDs) [5, 6] is done in order make formal analysis of the overall system as efficient as possible. But there are also theorem provers for multimodal or temporal logics available nowadays (see e.g. [21]). Let us now briefly introduce CTL*.

Definition 4.9

The syntax of CTL* formulae is given by the following rules. We begin with *state formulae*:

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 formulae, then $\neg f_0, f_1 \wedge f_2$ and $f_1 \vee f_2$ are also state formulae.
3. If g is a path formula, then $\mathbf{E}g$ and $\mathbf{A}g$ are state formulae.

1. $s \models p$ iff $p \in L(s)$.
2. $s \models \neg f$ iff $s \not\models f$.
3. $s \models f_1 \wedge f_2$ iff $s \models f_1$ and $s \models f_2$.
4. $s \models f_1 \vee f_2$ iff $s \models f_1$ or $s \models f_2$.
5. $s \models \mathbf{E}f$ iff there is a path π from s such that $\pi \models f$.
6. $s \models \mathbf{A}f$ iff for every path π starting from s , we have $\pi \models f$.
7. $\pi \models f$ iff $s \models f$ where s is the first state of π .
8. $\pi \models \neg g$ iff $\pi \not\models g$.
9. $\pi \models g_1 \wedge g_2$ iff $\pi \models g_1$ and $\pi \models g_2$.
10. $\pi \models g_1 \vee g_2$ iff $\pi \models g_1$ or $\pi \models g_2$.
11. $\pi \models \mathbf{X}g$ iff $\pi^2 \models g$.
12. $\pi \models \mathbf{F}g$ iff there exists a $n \geq 1$ such that $\pi^n \models g$.
13. $\pi \models \mathbf{G}g$ iff for all $n \geq 1$ we have $\pi^n \models g$.
14. $\pi \models g_1 \mathbf{U}g_2$ iff there exists an $n \geq 1$ such that $\pi^n \models g_2$ and for all $m < n$ it holds $\pi^m \models g_1$.

Figure 7: Semantics of CTL* formulae.

Two additional rules are needed in order to specify the syntax of *path formulae*:

1. If f is a state formula, then f is also a path formula.
2. If g_0, g_1 and g_2 are path formulae, then $\neg g_0, g_1 \wedge g_2$ and $g_1 \vee g_2, \mathbf{X}g_0, \mathbf{F}g_0, \mathbf{G}g_0$ and $g_1 \mathbf{U}g_2$ are path formulae.

Definition 4.10

A *path* in a Kripke structure is an infinite sequence of states, $\pi = s_1, s_2, \dots$ such that for every $i \geq 1$ we have $R(s_i, s_{i+1})$. We use π^i to denote the *suffix* of π starting at s_i . If g is a guard formula, the notation $s \models g$ means that g holds at state s . Analogously, $\pi \models g$ means that g holds along the path π .

4.4 Formal Analysis of System Properties

The relation \models is inductively defined as in Figure 7. There f, f_1 and f_2 denote state formulae, while g, g_1 and g_2 denote path formulae. We can express both planning as done in [14], 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, then we just have to check the formula $\mathbf{EF} \sigma$. The meaning of the temporal operators is as listed below. For more details, the reader is referred to [6].

1. \mathbf{E} (“for some computation path”) is an existential quantification.
2. \mathbf{A} (“for all computation paths”) is a universal quantification.
3. \mathbf{X} (“next time”) requires that a property holds in the second state of the path.
4. \mathbf{F} (“future”) specifies that a property will hold at some state of the path.
5. \mathbf{G} (“globally”) specifies that a property holds at every state on the path.

Example 4.11

We can now formulate temporal logic formulae checking certain properties, e.g. that from any state it is possible to get to a state where obstacles are avoided: $\mathbf{AG}(\mathbf{EF} \text{Avoid})$. This could also be seen as a planning task. We can also express the fact that without obstacle avoidance, the agent alternately steps and turns:

$$\mathbf{AG}(\neg \text{Avoid} \wedge \mathbf{AX} \neg \text{Avoid} \rightarrow (c = \text{Turn} \rightarrow \mathbf{AX} c = \text{Step}) \wedge (c = \text{Step} \rightarrow \mathbf{AX} c = \text{Turn}))$$

Thus, in summary, we have seen that model checking of single agent systems is possible (Example 4.11). Of course, these techniques can also be applied to general multiagent systems. For this, the system specification is translated into a Kripke structure. Unfortunately, the hierarchical structure of the original state machine is lost. But this seems to be necessary in order to exploit the efficient standard procedures available for model checking.

5 Final Remarks

5.1 Related Work

Since the first proposal of statecharts, there are a few attempts to capture their semantics in a rigorously formal manner. The paper [27] concentrates on the problem how steps of a state machine can be defined. First, a non-deterministic algorithm for computing all possible steps from a given configuration c is given. Second, a more declarative, equivalent definition based on solving a fixpoint equation is given. The distinction between micro- and macro-steps is present in [27], which we adapted for our purposes here (in Section 2.2).

Another semantics of statecharts is given in [13]. It describes the semantics as given in the STATEMATE system, which was the first executable semantics defined for the language. A basic step algorithm is provided, that disallows conflicts among different transitions starting from one state. However, in one step, transitions starting at different

states may occur. Here, outer transitions are preferred over inner ones. The advantage of this procedure is that the agents are more reactive. We adopted this implicit priority relation among states in the RoboLog system (see Section 3.4).

The development of formal semantics for statecharts is still a field of active research. For instance, [20] provides a so-called compositional semantics for statecharts and introduces a completely term-based notation for statecharts and transitions. [19] presents a semantics for statecharts by reduction to intuitionistic Kripke models, improving the definition in [27]. Last but not least, the very recent paper [16] states another interesting approach for defining a semantics for statecharts, namely based on graph transformations.

Many languages for controlling a robot, make use of finite state automata, e.g. *Colbert* [15]. Finite state automata can be viewed as a special case of the state machines considered here, where a notion of concurrency is present in addition. Implementations of agents relying on the subsumption architecture [3, 4] also follow the idea, that (robot) agents can be described by (finite) state machines. There, several independent finite state machines work in parallel, hosted in several layers, partly interacting with each other (e.g. suppression or inhibition). This means, that the concept of concurrency is heavily exploited in this framework.

Formalisms like the situation calculus (see e.g. [9, 17, 30]) also provide a means for specifying dynamic domains, e.g. with successor state axioms [28]. In essence, they allow us to specify the binary relation \rightsquigarrow between subsequent situations for each action a . The situation changes over time, i.e. the variables in V change their value in each step of the multiagent system. Therefore, also more complex properties of the whole system change. This is modeled in the *situation calculus* by so-called fluents. A *fluent* is a predicate describing a system property changing over time, it has a special (last) argument, namely the actual situation.

Example 5.1

Let us illustrate this with an example from the soccer domain (Example 1.6). As fluent we consider the property $close(d_{goal})$, i.e., the goal is close. Clearly, neither Turn nor a Kick action changes this property, whereas a Dash does, if the agent withdraws from the goal. This can be expressed by the following successor state axiom:

$$close(d_{goal}, \langle action | sit \rangle) \leftrightarrow \begin{array}{l} \exists p action = Dash(p) \wedge |a_{goal}| > 90^\circ \\ \vee (\exists a action = Turn(a) \vee \exists p action = Kick(p)) \wedge close(d_{goal}, sit) \end{array}$$

In the situation calculus, a situation is identified by a list of actions, namely the sequence of actions (in reverse order) that lead to the current situation, starting from the initial situation, represented by the empty list. Since there is no notion of concurrent modules or agents present in the situation calculus (but see [9, 12]), basically the calculus is merely appropriate for the description of a single-agent system where the effect of actions is deterministic. This certainly is an advantage and disadvantage at

the same time. Nevertheless, we take the view in this paper, that the effects of actions may be non-deterministic.

So far, we only considered related work on single agents. But there are also works on extending UML (statecharts) for the specification of multiagent systems. For instance, [2] presents an approach for agent-oriented software engineering by introducing a new level into the specification, called agent level. For this, several (new) types of diagrams are employed: ontology diagrams, architecture diagrams, protocol and role diagrams. This approach is different from the point of view taken here: we want to give a concise overview of the functionality of the complete multiagent system by means of only *one* type of diagram, namely statecharts.

[1] presents an approach for model checking multiagent systems. It also makes use of the temporal logics CTL, but there the multiagent system is given by a BDI architecture. Thus, agents are specified by some agent programs where temporal operators and belief operators are allowed. A multiagent model checking algorithm is given in [1], which takes two arguments namely a view of the multiagent system and a formula, that we want to model check. Thus the main difference to our approach is that we specify agents with statecharts. We think this is an advantage, because this allows us a quite high-level design of multiagent systems.

5.2 Conclusions

In this paper, we presented a method from the visual specification to the formal verification and implementation of multiagent systems. We integrated different methods from the field of artificial intelligence such as qualitative (spatial) reasoning and the situation calculus and showed the relevance to the field of multiagent systems. While the development of a running system for formal verification of multiagent systems is subject to future work, the procedure for the formal specification of multiagent systems presented in this paper has already been applied in the RoboLog system, a multiagent system for simulated robotic soccer. The team RoboLog Koblenz participated successfully in the RoboCup world championships in 1999 and 2000, and will participate in the competition in 2001.

Acknowledgments

I would like to thank Jan Murray and Toshiaki Arai for helpful discussions about the topic of this paper.

References

- [1] M. Benerecetti, F. Giunchiglia, and L. Serafini. A model checking algorithm for multi-agent systems. In J. P. Müller, M. P. Singh, and A. S. Rao, editors,

- Proceedings of 5th International Workshop on Intelligent Agents 1998: Agents Theories, Architectures, and Languages*, LNAI 1555, pages 163–176. Springer, Berlin, Heidelberg, New York, 1999.
- [2] F. Bergenti and A. Poggi. Exploiting UML in the design of multi-agent systems. In A. Omicidi, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World*, LNCS 1972, pages 106–113. Springer, Berlin, Heidelberg, New York, 2000.
 - [3] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
 - [4] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991.
 - [5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
 - [6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, London, 1999.
 - [7] E. Clementini, P. Di Felice, and D. Hernández. Qualitative representation of positional information. *Artificial Intelligence*, 95(2):317–356, 1997.
 - [8] E. Corten, K. Dorer, F. Heintz, K. Kostiadis, J. Kummeneje, H. Myritz, I. Noda, J. Riekkki, P. Riley, P. Stone, and T. Yeap. *Soccerserver Manual (for Soccerserver Version 5.00 and later)*, 5th edition, 1999.
 - [9] G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In M. E. Pollack, editor, *Proceedings of 15th International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, 1997. IJCAI Inc., San Mateo, CA, Morgan Kaufmann, Los Altos, CA. Volume 2.
 - [10] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers. Volume B.
 - [11] J. H. Gallier. *Logic for Computer Science*. John Wiley & Sons, New York, Chicester, Brisbane, 1987.
 - [12] H. Grosskreutz and G. Lakemeyer. Towards more realistic logic-based robot controllers in the GOLOG framework. *KI*, 4/00:11–15, 2000.
 - [13] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

- [14] R. M. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.
- [15] K. Konolige. COLBERT: A language for reactive control in Sapphira. In G. Brewka, C. Habel, and B. Nebel, editors, *KI-97: Advances in Artificial Intelligence – Proceedings of the 21st Annual German Conference on Artificial Intelligence*, LNAI 1303, pages 31–52, Freiburg, 1997. Springer, Berlin, Heidelberg, New York.
- [16] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In *Conference on UML*, 2001. To appear.
- [17] 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.
- [18] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, Heidelberg, New York, 1987.
- [19] G. Lüttgen and M. Mendler. Fully-abstract statechart semantics via intuitionistic kripke models. In U. Montanari, J. D. Rolim, and E. Welzl, editors, *Proceedings of 27th International Colloquium on Automata, Languages and Programming*, LNCS 1853, pages 163–174. Springer, Berlin, Heidelberg, New York, 2000.
- [20] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *Proceedings of ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering*, pages 120–129, San Diego, CA, 2000. ACM Press. ACM Software Engineering Notes 25(6).
- [21] F. Massacci and F. M. Donini. Design and results of TANCS-2000 non-classical (modal) systems comparison. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (Proceedings of TABLEAUX-2000)*, LNAI 1847, pages 52–56. Springer, Berlin, Heidelberg, New York, 2000.
- [22] J. Murray. Soccer agents think in UML. Diplomarbeit D 610, Fachbereich Informatik, Universität Koblenz-Landau, 2001.
- [23] J. Murray, O. Obst, and F. Stolzenburg. RoboLog Koblenz 2000. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer WorldCup IV*, LNAI 2019, pages 469–472. Springer, Berlin, Heidelberg, New York, 2001. Team description.
- [24] 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 WorldCup IV*, LNAI 2019, pages 199–208. Springer, Berlin, Heidelberg, New York, 2001.

- [25] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, 1999. Version 1.3, June 1999.
- [26] 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 (AAAI 2000)*, pages 3–17, Austin, Texas, 2000.
- [27] 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, Sendai, Japan, 1991. Springer, Berlin, Heidelberg, New York.
- [28] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, New York, 1991.
- [29] J. Renegar. On the computational complexity and geometry of the first order theory of the reals I-III. *Journal of Symbolic Computation*, 13(3):255–352, 1992.
- [30] S. Russell and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.
- [31] F. Stolzenburg, O. Obst, J. Murray, and B. Bremer. Spatial agents implemented in a logical expressible language. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, LNAI 1856, pages 481–494. Springer, Berlin, Heidelberg, New York, 2000.
- [32] A. Tarski. *A decision method for elementary algebra and geometry*. University of California Press, Berkeley, Los Angeles, 1951.
- [33] K. Zimmermann and C. Freksa. Qualitative spatial reasoning using orientation, distance, and path knowledge. *Applied Intelligence*, 6:49–58, 1996.

Available Research Reports (since 1998):

2001

- 6/2001** *Frieder Stolzenburg*. From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking.
- 5/2001** *Oliver Obst*. Specifying Rational Agents with Statecharts and Utility Functions.
- 4/2001** *Torsten Gipp, Jürgen Ebert*. Conceptual Modelling and Web Site Generation using Graph Technology.
- 3/2001** *Carlos I. Chesñevar, Jürgen Dix, Frieder Stolzenburg, Guillermo R. Simari*. Relating Defeasible and Normal Logic Programming through Transformation Properties.
- 2/2001** *Carola Lange, Harry M. Sneed, Andreas Winter*. Applying GUPRO to GEOS – A Case Study.
- 1/2001** *Pascal von Hutten, Stephan Philippi*. Modelling a concurrent ray-tracing algorithm using object-oriented Petri-Nets.

2000

- 8/2000** *Jürgen Ebert, Bernt Kullbach, Franz Lehner (Hrsg.)*. 2. Workshop Software Reengineering (Bad Honnef, 11./12. Mai 2000).
- 7/2000** *Stephan Philippi*. AWPN 2000 - 7. Workshop Algorithmen und Werkzeuge für Petrinetze, Koblenz, 02.-03. Oktober 2000 .
- 6/2000** *Jan Murray, Oliver Obst, Frieder Stolzenburg*. Towards a Logical Approach for Soccer Agents Engineering.
- 5/2000** *Peter Baumgartner, Hantao Zhang (Eds.)*. FTP 2000 – Third International Workshop on First-Order Theorem Proving, St Andrews, Scotland, July 2000.
- 4/2000** *Frieder Stolzenburg, Alejandro J. García, Carlos I. Chesñevar, Guillermo R. Simari*. Introducing Generalized Specificity in Logic Programming.
- 3/2000** *Ingar Uhe, Manfred Rosendahl*. Specification of Symbols and Implementation of Their Constraints in JKogge.
- 2/2000** *Peter Baumgartner, Fabio Massacci*. The Taming of the (X)OR.
- 1/2000** *Richard C. Holt, Andreas Winter, Andy Schürr*. GXL: Towards a Standard Exchange Format.

1999

- 10/99** *Jürgen Ebert, Luuk Groenewegen, Roger Süttenbach*. A Formalization of SOCCA.
- 9/99** *Hassan Diab, Ulrich Furbach, Hassan Tabbara*. On the Use of Fuzzy Techniques in Cache Memory Management.
- 8/99** *Jens Woch, Friedbert Widmann*. Implementation of a Schema-TAG-Parser.
- 7/99** *Jürgen Ebert, and Bernt Kullbach, Franz Lehner (Hrsg.)*. Workshop Software-Reengineering (Bad Honnef, 27./28. Mai 1999).
- 6/99** *Peter Baumgartner, Michael Kühn*. Abductive Coreference by Model Construction.
- 5/99** *Jürgen Ebert, Bernt Kullbach, Andreas Winter*. GraX – An Interchange Format for Reengineering Tools.
- 4/99** *Frieder Stolzenburg, Oliver Obst, Jan Murray, Björn Bremer*. Spatial Agents Implemented in a Logical Expressible Language.
- 3/99** *Kurt Lautenbach, Carlo Simon*. Erweiterte Zeitstempelnetze zur Modellierung hybrider Systeme.
- 2/99** *Frieder Stolzenburg*. Loop-Detection in Hyper-Tableaux by Powerful Model Generation.
- 1/99** *Peter Baumgartner, J.D. Horton, Bruce Spencer*. Merge Path Improvements for Minimal Model Hyper Tableaux.

1998

- 24/98** *Jürgen Ebert, Roger Süttenbach, Ingar Uhe*. Meta-CASE Worldwide.
- 23/98** *Peter Baumgartner, Norbert Eisinger, Ulrich Furbach*. A Confluent Connection Calculus.
- 22/98** *Bernt Kullbach, Andreas Winter*. Querying as an Enabling Technology in Software Reengineering.
- 21/98** *Jürgen Dix, V.S. Subrahmanian, George Pick*. Meta-Agent Programs.
- 20/98** *Jürgen Dix, Ulrich Furbach, Ilkka Niemelä*. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations.
- 19/98** *Jürgen Dix, Steffen Hölldobler*. Inference Mechanisms in Knowledge-Based Systems: Theory and Applications (Proceedings of WS at KI '98).

- 18/98** *Jose Arrazola, Jürgen Dix, Mauricio Osorio, Claudia Zepeda.* Well-behaved semantics for Logic Programming.
- 17/98** *Stefan Brass, Jürgen Dix, Teodor C. Przymusiński.* Super Logic Programs.
- 16/98** *Jürgen Dix.* The Logic Programming Paradigm.
- 15/98** *Stefan Brass, Jürgen Dix, Burkhard Freitag, Ulrich Zukowski.* Transformation-Based Bottom-Up Computation of the Well-Founded Model.
- 14/98** *Manfred Kamp.* GReQL – Eine Anfragesprache für das GUPRO-Repository – Sprachbeschreibung (Version 1.2).
- 12/98** *Peter Dahm, Jürgen Ebert, Angelika Franzke, Manfred Kamp, Andreas Winter.* TGraphen und EER-Schemata – formale Grundlagen.
- 11/98** *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.
- 10/98** *Jörg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.
- 9/98** *Dieter Zöbel.* Schedulability criteria for age constraint processes in hard real-time systems.
- 8/98** *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.
- 7/98** *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.
- 6/98** *Ulrich Furbach, Michael Kühn, Frieder Stolzenburg.* Model-Guided Proof Debugging.
- 5/98** *Peter Baumgartner, Dorothea Schäfer.* Model Elimination with Simplification and its Application to Software Verification.
- 4/98** *Bernt Kullbach, Andreas Winter, Peter Dahm, Jürgen Ebert.* Program Comprehension in Multi-Language Systems.
- 3/98** *Jürgen Dix, Jorge Lobo.* Logic Programming and Nonmonotonic Reasoning.
- 2/98** *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme.* Zeitstempelnetze in technischen Anwendungen.
- 1/98** *Manfred Kamp.* Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.