

UNIVERSITÄT
KOBLENZ · LANDAU



**Multiagent Systems Specification by
UML Statecharts Aiming at
Intelligent Manufacturing**

Toshiaki Arai, Frieder Stolzenburg

12/2001



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/>

Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing*

Toshiaki Arai ara@mmc.co.jp Knowledge Industry Dept. Advanced Systems Center Mitsubishi Materials Corporation Koishikawadaikoku Bldg. 1-3-25 Koishikawa, Bunkyo-Ku Tokyo 112-0002 JAPAN	Frieder Stolzenburg stolzen@uni-koblenz.de Institut für Informatik Fachbereich Informatik Universität Koblenz-Landau Rheinau 1 56075 Koblenz GERMANY
---	---

Abstract

Multiagent systems are a promising new paradigm in computing, which are contributing to various fields. Many theories and technologies have been developed in order to design and specify multiagent systems, however, no standard procedure is used at present. Industrial applications often have a complex structure and need plenty of working resources. They require a standard specification method as well. As the standard method to design and specify software systems, we believe that one of the key words is simplicity for their wide acceptance. In this paper, we propose a method to specify multiagent systems, namely with UML statecharts. We use them for specifying almost all aspects of multiagent systems, because we think that it is an advantage to keep everything in one type of diagram.

We apply our method to different domains, namely to robotic soccer and a network application. This approach enables not only standardized design of multiagent systems, but also almost automatic translation of the specification into a running implementation (here: into Prolog). Moreover, the verification or formal analysis is feasible, because of the rigidly formal manner of the system specification. We concentrate

*This paper emerged while the first author was visiting the University in Koblenz, Germany, from October 2000 to September 2001.

on the formal specification of multiagent systems in general and its application to robotic soccer, which is already implemented, and to networking. The application to different domains—with homogeneous or heterogeneous agents—corroborates the generality of the proposed approach.

Keywords: agent-based software engineering; methodologies for specification, design, implementation, and validation; standards for agents and multiagent systems; unified modeling language (UML).

1 Introduction

Almost all software systems can be viewed as multiagent systems. They provide a good means for a robust architecture of software systems. The whole system can work, even if parts of them are out of function. Thus, multiagent systems are interesting for industrial applications and also for research. They are one of the foremost research topics in the field of artificial intelligence. Therefore, in this paper, we present a way to specify and implement such systems in a systematical and formally clean way, and show how this can be applied to robotic soccer and to internet-based activities in an industrial application.

Internet-based activities such as web services and peer-to-peer (P2P) applications are drawing more and more attention, because they are feasible to enhance computing-resource sharing, create new markets, and so forth. Nowadays internetworking technology is getting more and more important in this context. With respect to multiagent systems, they are widely accepted as the best method to design network architecture, since they enable distributed, flexible and robust systems. Many industrial applications have been proposed. For example, advanced manufacturing can be designed as a multiagent system. For an overview on applications of agent technology see [5].

1.1 Motivation

In general, a multiagent system is a system in which several interacting, intelligent agents pursue some set of goals or tasks. An agent is a computational entity such as a software program or a robot that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behavior is at least partially depends on its own experience [14]. An important question, of course, is how can such systems of agents be designed and implemented? We think that current software specification techniques must be adapted with respect to multiagent systems, such that coordination of several agents can be expressed.

In this paper, we propose a method that makes use of only one type of diagram for the specification of multiagent systems, namely UML statecharts [9]. The advantage of this procedure is that on the one hand designers and programmers can rapidly develop this system, and on the other hand this specification can easily be turned into executable code, because we have to consider only one type of diagram. Finally, the formalism based on the statecharts should also allow for the verification or formal analysis of multiagent systems, e.g., by model checking (see also [1, 12]).

Therefore, in this paper, the focus is laid on a rigidly formal specification method of multiagent systems, because this must be the basis for more formal analyses of such systems. In addition, we will demonstrate the applicability of the proposed approach in different applications (namely robotic soccer and internetworking). For more details on how almost all features of UML statecharts can be exploited for the development of multiagent systems and the translation of such a specification into executable code, the interested reader is referred to [6, 7].

1.2 Overview of the Paper

One can distinguish between systems of homogeneous agents, where all agents in principle have the same capabilities, and systems of heterogeneous agents, where different agents have different abilities performing different tasks. For example, teams of (robotic) soccer agents can be viewed as a system of (more or less) homogeneous agents, whereas the above-mentioned web services and peer to peer applications are appropriate for a system of heterogeneous agents. In the following, we will consider both applications in more detail: simulated robotic soccer in Section 3.1, and the network application in Section 3.2. We show that both kinds of multiagent systems can be specified with one and the same formalism, namely statecharts, which we introduce beforehand in Section 2.

Multiagent systems are studied in the domain of robotic soccer, in which the behavior of agents including collaboration is specified by means of UML statecharts [8]. Since this method seems to be applicable to the above industrial applications, we introduce the network application domain in order to investigate the effectiveness of the method in this domain. Then, we discuss the general problem of specifying multiagent systems by means of UML statecharts (Section 4) and related works (Section 5), before we come to the conclusions after that (Section 6).

2 State Machines

Dynamic systems can be described appropriately as state machines. Therefore, statecharts are used quite often for the specification of dynamic or procedural aspects of software systems, also in industrial applications. In order to provide a better understanding of statecharts, we give their formal definition in the following. Later (in Sections 3 and 4) we will demonstrate that statecharts are a good means not only for dynamic systems but also for multiagent systems, where several independent agents interact.

2.1 Basic Components

Statecharts are a part of UML [9] and a well accepted means to specify dynamic behavior of software systems. They can be described in a rigorously formal manner, allowing for flexible specification, implementation and analysis of multiagent systems [10, 12]. In statecharts, states are connected via transitions with conditions and actions annotated. They are represented as rectangles with round corners and can be structured hierarchically. Transitions are drawn as directed arcs that interconnect the states. Transitions are annotated by events (e.g. reception of a request or change of some value), guards (which are conditions), and actions. Since states may be simple, composite or concurrent, the behavior of agents or their state machines, respectively, cannot be described by sequences of simple states (as for finite automata), but of configurations. For more details, the reader is referred to [9] and Section 2.2.

The state machine in Figure 1 sketches the overall behavior of robotic soccer agents (which we will discuss in more detail in Section 3.1). 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 is Behave, whose initial state is Init. The three states Init, Attack and Defend belong to the main (start) state Behave. Its initial state Init permits a transition annotated with KickOff/Kick(100%) (with empty guard 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, it is in the Attack state; otherwise it is in Defend state. Furthermore, there are transitions from Attack and Defend back to Init, if somebody scores a goal.

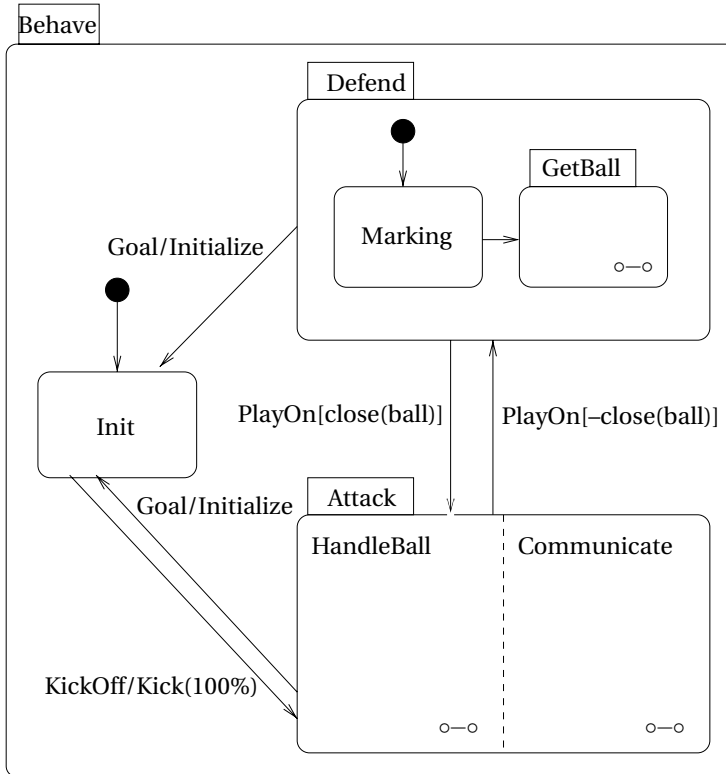


Figure 1: State machine for overall behavior.

2.2 Configurations and Steps

How can the behavior of state machines be described? For this, we now introduce the concepts of configuration and step.

Definition 1 (Configuration) *A configuration c is a rooted tree of states, where the root node is the topmost initial state of the overall state machine. A configuration must be completed by the following procedure: if there is a leaf node in c labeled with a composite state s , then the initial state of s is introduced as immediate successor of s ; if there is a leaf node in c labeled with a concurrent state s , then the tree branches at this point.*

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

Agents perform steps from one configuration to another. For proper multi-agent systems, we also must take into account that several agents or different components of one and the same agent may perform steps at the same time. Therefore, one may distinguish between micro- and macro-steps as in [11, 12]. The right part of Figure 2 shows the configuration of the state machine after one transition from Init to Attack. For more details, the reader is referred to [11, 12]. We will here only briefly summarize the definitions.

Definition 2 (Micro-step) *A micro-step from one configuration c of a state machine to another configuration c' by means of some transition t from state s to state s' with annotation $\text{event}[\text{guard}]/\text{action}$ in the current situation—written $c \rightarrow_t c'$ —is possible iff:*

1. c contains a node labeled with s ,
2. c' is identical with c except that s together with its subtree in c is replaced by the completion of s' , and
3. the annotated event and guard holds in the actual situation.

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 the definition. 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 $\circ-\circ$ in it.

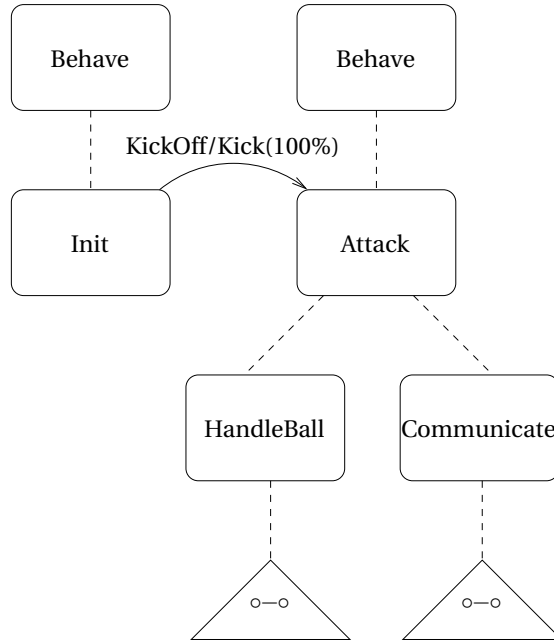


Figure 2: Configuration after one transition.

Definition 3 (Macro-step) A macro-step from the configuration c to a configuration c' with given micro-steps is possible iff:

1. all micro-steps are possible in c ,
2. all 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 .

With macro-steps, we are able to model concurrent behavior. This is very helpful for the description of multiagent systems, since there we have several entities which interact concurrently. We will heavily make use of concurrent states in our example applications (see Sections 3 and 4). Concurrent states are the prerequisite for proper macro-steps, i.e. where several micro-steps are executed in parallel.

3 Example Applications

In this section, we will introduce the two applications robotic soccer and networking in a bit more detail. Robotic soccer (see Section 3.1) has been tackled mainly by the second author, while the first author considered the networking application (see Section 3.2). The interesting thing is that, although both

applications look rather different at first glance, we can apply the same techniques to describe them as a multiagent system. We will do this in this section and continue in Section 4, where we address the problem of specifying the interaction of several, possibly heterogeneous agents.

3.1 Robotic Soccer

In the RoboCup initiative, the soccer game is chosen as a central topic of research, aiming at innovations to be applied for socially significant problems and in industry. The *RoboCup* is an international joint project to promote artificial intelligence, robotics, and related fields. In order to perform actually a soccer game for a robot team, various technologies must be incorporated including design principles of autonomous agents, multiagent collaboration, strategy acquisition (see e.g. [4]), real-time reasoning etc. The RoboCup consists of several leagues with real robots in different sizes or virtual, i.e. simulated robots. The simulation league offers a software platform for research on the software aspects of RoboCup.

In our context, the software design aspect is the most important one. Soccer agents can be designed with a three-level approach (see Figure 3) [6, 12]: the *mode level* contains the most abstract desires an agent has (e.g. setup, attack, defend); the *script level* provides plan skeletons that are used as long as the mode is not changed (e.g. marking, passing, role exchange); the *skill level* hosts basic actions (e.g. kick, dribble). In the soccer domain, usually all agents have an identical internal structure except for the goal-keeper; all other agents are homogeneous. Therefore, it is quite natural to state the behavior of the agents within one state machine. But we will see that even for heterogenous agent systems the whole multiagent system can be specified within one statechart (see Section 3.2).

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 [8, 13], 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 can be programmed conveniently in Prolog, implementing decision trees. In addition, by this procedure, agent programmers have a full programming language available and can design and implement agents without many restrictions.

In summary, the soccer domain is well-suited for homogeneous agents. Each agent can be described by one and the same state machine. Nevertheless, agents can take different roles depending on the respective situation they are in, e.g. in a double passing situation. Interaction among agents is triggered by synchronization states. All this can be specified conveniently

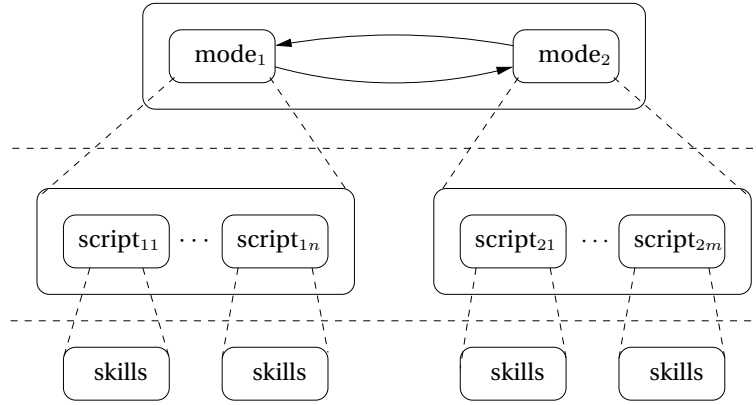


Figure 3: Layered architecture.

by statecharts. Their hierarchical structure makes a modular specification of multiagent systems possible.

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, 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 4.

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 4, the left bar is a *fork state*, while the right bar 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 ranging from 0 to 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 4, 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.

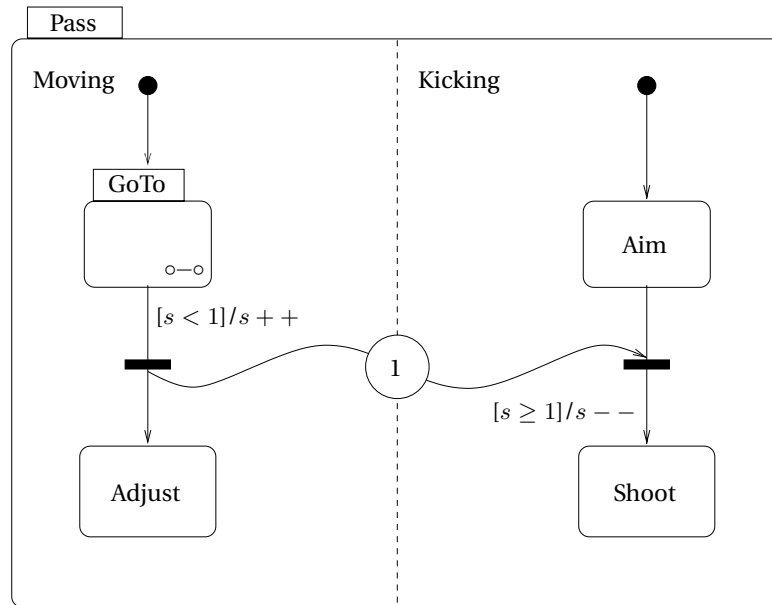


Figure 4: Machine with concurrent states for Passing.

3.2 Network Application

Nowadays, internet technologies are widespread in industrial applications, and plenty of data are exchanged via the internet. Such data include HTML page contents, database tables, numerous plant operation data, etc. In this context, data exchange via the internet involves some problems such as security, communication protocol among agents, internetworking technology. The peer-to-peer application is an example of such technology. Our first approach for the design of the scheme for data exchange is to consider it as a multiagent system with its behavior, using the same method as for robotic soccer.

Let us now introduce the peer-to-peer network application domain as the second example of a multiagent system in which concurrent actions and the collaboration among several agents are explicitly specified. Recently the internet and networks in general prevail widely, accompanied with advanced technologies, which are able to induce the development of new industrial applications. The integration of manufacturing resources can be achieved by means of such applications based on multiagent systems. For instance, the peer-to-peer network application is expected to play an important role in a real intelligent manufacturing system, whose components such as facilities and databases are distributed. We show how an industrial multiagent system is designed using one and the same method as for robotic soccer, where various technologies are investigated such as design principles of autonomous

agency, multiagent collaboration, strategy acquisition, etc.

The overall design of the multiagent system is shown in Figure 5. In this domain, we consider heterogeneous agents, namely: user agent, broker agent, proxy agent and sensor agent, with the following behavior. User agents communicate with each other, i.e., they send or receive files via the internet, supported by the broker agent and the other agents. This primary function will be invoked by an external event invoked by the user. The broker agent allocates the destination of each file among the clients, i.e. the user agents. Thus, the broker agent enables a file-exchange function. The broker agent and the user agent cannot directly communicate beyond the firewall, therefore the proxy agent is introduced. The sensor agent will diagnose network condition and notify the results to user agents so as to optimize the user agents' behavior.

In summary, although we have heterogeneous agents with different functionality in this application, the multiagent systems again can be specified by means of one statechart also in this case. Different actions which are executed in parallel can be expressed by several regions in the statechart. We will consider this now in more detail in the next Section 4.

4 Specifying Multiagent Systems

In this section, we will show how the behavior of single or several agents are specified by means of statecharts, in the domains of both robotic soccer and network application. Many features in multiagent systems can be represented by the same method. It seems to be remarkable that even in different domains, i.e., in both robotic soccer and network application domains, the same method is feasible to specify important aspects of multiagent systems, although both domains have different characteristics and function. This approach enables not only the design of multiagent systems, but also the verification and formal analysis of the system should be allowed by the strict formalism in the long run (see also [12]).

4.1 The Switch of Roles

As shown in Figure 6, the user agent can be designed as a state machine, whose behavior is similar to the soccer player agent in Figure 1. At the top-level (under the Behave state), the user agent has the following sub-states: Init, Wait, Put and Get. These top-level states and sub-states correspond to the mode level and the script level within the three-level approach in the robotic soccer domain (in Figure 3). Thus, once again we can exploit the hierarchical structure of statecharts.

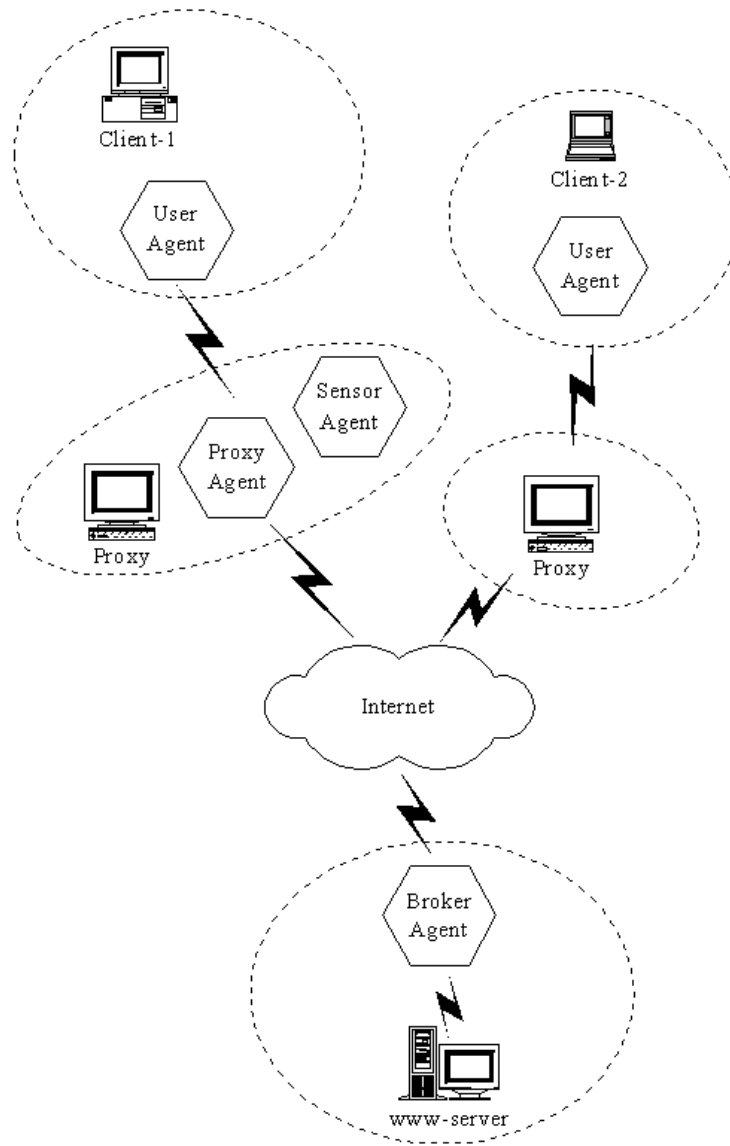


Figure 5: Overall architecture of the network application.

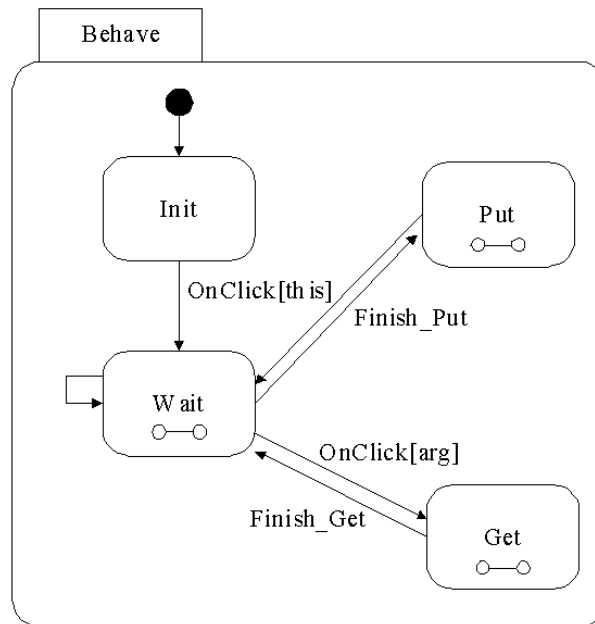


Figure 6: State machine for the user agent.

According to external events, a user agent is able to provide one of two different functions, Put or Get. Thus, it is assumed that a user agent has two roles: Put and Get, which is analogical with the Attack and Defend states in the robotic soccer domain. In the Put state, the user agent transfers a file to another agent, while in the Get state the agent receives the file. Both actions are initiated by a mouse click event. We discuss the Put state in greater detail in the following (see also Figures 7 and 8).

4.2 Synchronization

The Put state is a composite state and has two concurrent sub-states, Send and VerifySend, as depicted in Figure 7. The Send sub-machine is designed to provide the file transfer function, while the VerifySend sub-machine is introduced in order to ensure the function. The synchronization of these two concurrent states is enabled by synch states in UML, which are rendered as circles between two concurrent regions in statecharts. A synch state is associated with fork and join states, which are indicated as thick bars and stand for the transition between concurrent states. Synch states enable synchronization as a consequence.

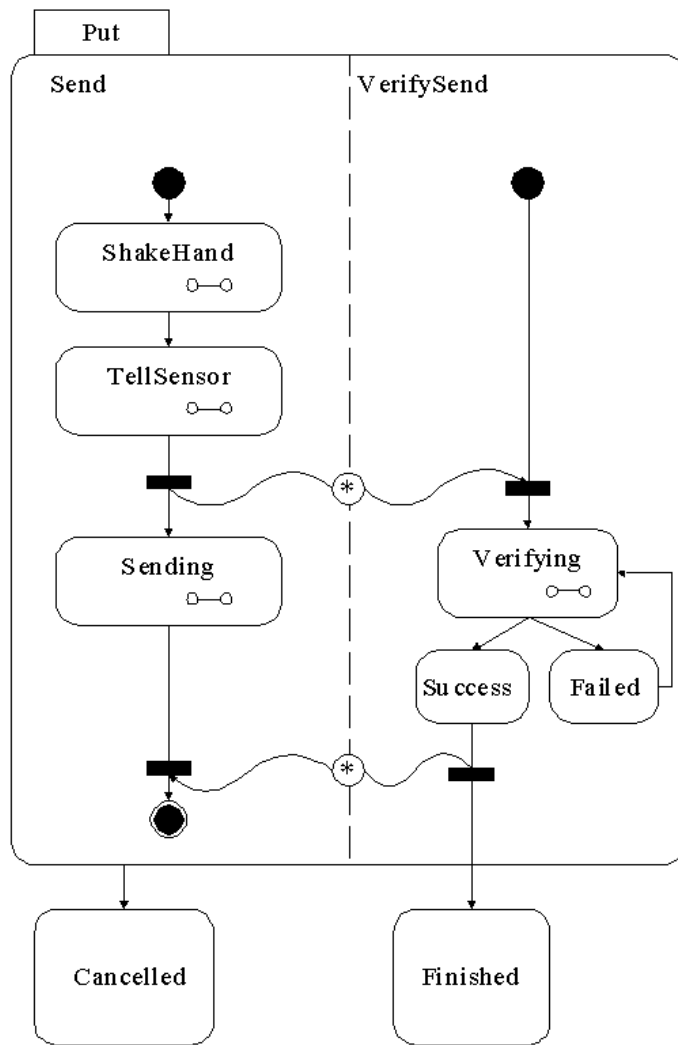


Figure 7: Synchronization of agents.

4.3 Collaboration

Statecharts can represent the collaboration not only within one agent, but also the collaboration among heterogeneous agents. Figure 8 illustrates three concurrent regions, whose rightmost region belongs to a sensor agent, while the others are in a user agent. In Figure 8, the CheckFile sub-state is designed to perform the checking task for a transferred file, and the VerifySend sub-state is a kind of mirroring of CheckFile in order to enable fast file transfer. This example indicates that a synch state can represent collaboration among agents as well as concurrency within a single agent.

In other words, both interaction of several agents and parallel threads or activities of one agent can be expressed by synch states. State machines with synch states can be translated into simpler ones without synch states, after introducing dedicated synchronization variables (as e.g. in Figure 4). While transforming concurrent regions into executable code, the possibility of macro-steps has to be taken into account (see Definition 3). In practice, the actions of agents are coordinated by external events or variables for synchronization. All this is realized by a state machine, coded explicitly in Prolog.

5 Related Works

In this paper, we propose a method for the specification of multiagent systems that makes use of standard software engineering methods and is formal enough that we can easily achieve an executable specification, and formal system analysis and verification is possible. The combination of these two features seems to be rather original. In the following, we discuss works extending the Unified Modeling Language (UML) for agents (Section 5.1) and works on the analysis of agent systems which are in most cases logic-based (Section 5.2). For an overview of applications of multiagent systems, e.g. in manufacturing, process control, telecommunication, information management, electronic commerce, games, medical applications, etc. the reader is referred to [5].

5.1 Agent Design with Extensions of UML

There is one proposal of the Object Management Group, which develops the UML standard, for extending UML for agents [10]. There, it is proposed to use a bunch of diagrams for expressing all aspects of multiagent systems, e.g. template and package diagrams, sequence and collaboration diagrams, and state and activity diagrams. The most important new aspect coming with systems of several agents is certainly interaction (communication, collaboration

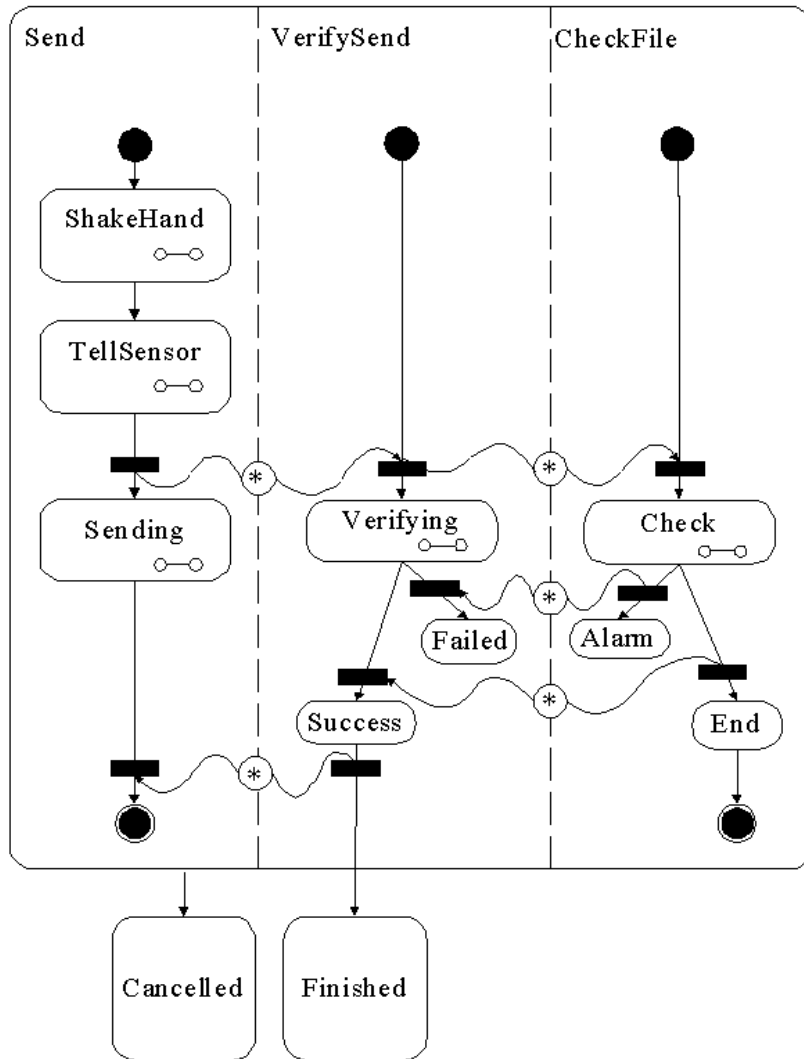


Figure 8: Collaboration among agents.

etc.). Therefore, it is stressed in [10] that interaction protocols can be specified in more detail (i.e. leveled or nested) using a combination of diagrams, e.g. with sequence or state diagrams. It is certainly a good idea to put everything in one (possibly nested or leveled) presentation. Another interesting idea—presented in [10]—is that statecharts can be labeled with the agents who are performing the respective action. However, because of the use of different types of diagrams, a direct transformation of the specification does not seem to be available for this approach, in contrast to the approach presented here.

According to [2], Software engineering describes a system at different levels of abstraction. Agent engineering introduces another level of abstraction: the agent level. As the previous approach discussed here, [2] also tries to exploit UML in the design of multiagent systems. It makes use of standard diagrams and languages such as class diagrams for ontologies and communication languages such as FIPA ACL or KQML. But they propose additional types of diagrams: architecture diagrams which are composed of a set of agent classes and relations among them. In addition, protocol and role diagrams are introduced. They are related to interaction protocols in UML. Thus, in summary, [2] follows the same idea as here, namely making use of standard notations as much as possible. However, the authors of [2] design multiagent systems by means of more than one diagram type.

5.2 Logic-based Agent Design and Analysis

[3] proposes an approach for the compositional verification of multiagent systems. Although here agents are specified in a hierarchically manner, no standard notation such as statecharts or class diagrams is used. Instead, a logic-based approach is used with a temporal multi-epistemic logic. Compositional verification for one abstraction level is based on assumptions, e.g., that the agent system consists of a number of agents and external world components, and a formal description exists how the whole multiagent system is composed (composition relation). This approach is very nice from a theoretical point of view. However, there are no tools available for this approach – neither for computer-aided software engineering nor for verification with the specific logic, because no standard procedures are used, although a graphical notation similar to UML collaboration diagrams is used.

In [1], a model-checking based decision procedure for multiagent systems is defined, which employs tools and technologies developed for model checking. A class of logics obtained by branching-time temporal logics (Computational Tree Logic CTL) and belief-desire-intention modal logics is used to specify systems of several agents. For this purpose, multiagent finite state machines are introduced, which allow a modular and incremental design

and analysis of multiagent systems. However, again there is no standard software engineering remedy used in this approach.

The approach in [12] shows a way from the specification of multiagent systems by statecharts to their formal analysis by model checking. Thus, for both the design and the analysis already known techniques are used. This seems to be an important aspect, if one wants the industry to accept a new approach. Therefore, [12] combines software engineering and artificial intelligence aspects. This is also the major emphasis of this paper.

6 Final Remarks

In this paper, we presented a method for the specification of multiagent systems. It can be applied in different fields, e.g. robotic soccer with homogeneous agents and in networking applications where we have a system of heterogeneous agents as presented in this paper. The advantage of the approach presented here is that the whole multiagent system can be specified with one type of diagram, namely statecharts. They can be translated more or less automatically into executable code, e.g. into Prolog.

Communication among agents can also be specified within statecharts. But here a more explicit representation seems to be desirable. This is subject of further work. Further work shall also concentrate on refining the formal specification of multiagent systems such that automatic analysis of certain system properties can be done. In addition, it certainly would be interesting that a formally verified multiagent specification is translated into running C code or any other programming language. The approach presented here lays the basis for the formal specification and verification of multiagent systems (see also [12]).

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] J. Engelfriet, C. M. Jonker, and J. Treur. Compositional verification of multi-agent systems in temporal multi-epistemic logic. *Journal of Logic, Language and Information*, 2001. To appear.
- [4] J. F. Grefenstette. Strategy acquisition with genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 14, pages 186–201. Van Nostrand Reinhold, New York, 1991.
- [5] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):275–306, 1998.
- [6] J. Murray. Soccer agents think in UML. Diplomarbeit D 610, Fachbereich Informatik, Universität Koblenz-Landau, 2001.
- [7] J. Murray. Specifying agents with UML in robotic soccer. Submitted, 2001.
- [8] J. Murray, O. Obst, and F. Stolzenburg. Towards a logical approach for soccer agents engineering. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, LNAI 2019, pages 199–208. Springer, Berlin, Heidelberg, New York, 2001.
- [9] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, September 2001. Version 1.4.
- [10] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop (AOIS) at the 17th National Conference on Artificial Intelligence*, pages 3–17, Austin, Texas, 2000.
- [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, Sendai, Japan, 1991. Springer, Berlin, Heidelberg, New York.
- [12] F. Stolzenburg. Reasoning about cognitive robotics systems. In R. Moratz and B. Nebel, editors, *Themenkolloquium Kognitive Robotik und Raumrepräsentation des DFG-Schwerpunktprogramms Raumkognition*, Hamburg, 2001.
- [13] 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.
- [14] G. Weiss, editor. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, London, 1999.

Available Research Reports (since 1998):

2001

- 12/2001** *Toshiaki Arai, Frieder Stolzenburg.* Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing.
- 11/2001** *Kurt Lautenbach.* Reproducibility of the Empty Marking.
- 10/2001** *Jan Murray.* Specifying Agents with UML in Robotic Soccer.
- 9/2001** *Andreas Winter.* Exchanging Graphs with GXL.
- 8/2001** *Marianne Valerius, Anna Simon.* Slicing Book Technology — eine neue Technik für eine neue Lehre?.
- 7/2001** *Bernt Kullbach, Volker Riediger.* Folding: An Approach to Enable Program Understanding of Preprocessed Languages.
- 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.