Modeling and Simulation of Tests for Agents

Martina Gierke, Jan Himmelspach, Mathias Röhl, and Adelinde M. Uhrmacher

University of Rostock
Institute of Computer Science
Albert-Einstein-Str. 21, D-18059 Rostock, Germany
{mroehl, gie, jh194, lin}@informatik.uni-rostock.de

Abstract. Software systems that are intended to work autonomously in complex, dynamic environments should undergo extensive testing. Model-based testing advocates the use of purpose-driven abstractions for designing appropriate tests. The type of the software, the objective of testing, and the stage of the development process influence the suitability of tests. Simulation techniques based on formal modeling concepts can make these abstractions explicit and operational. A simulation model is presented that facilitates testing of autonomous software within dynamic environments in a flexible manner. The approach is illustrated based on the application Autominder.

1 Introduction

Developing agents essentially means developing software that is able to successfully accomplish specified tasks in an environment that changes over time. This process is of an intrinsically experimental and exploratory nature. Little work has been done on developing methods for testing agents so far [1]. Current methodologies and tools that support agent design and implementation leave a gap between specifications and implementations [2].

In the following we will give a short overview about testing of agents. A discrete-event approach constitutes the basis for developing an experimental frame for testing agents. Various models with different roles in testing will be defined. Among those the interface between the software under test and the simulation system plays a central role and will be discussed in more detail. The setup of a concrete experimental frame will be illustrated on the application Autominder. The paper concludes with a discussion of related work.

2 Testing Agents

Different strategies for designing tests are used. Typically, black box and white box testing strategies are distinguished [3]. Whereas black box testing is concerned with examining the observable behavior of an implementation and focuses on functional requirements, white box testing makes use of knowledge about the internal structure of a program and how results are achieved.

K. Fischer et al. (Eds.): MATES 2006, LNAI 4196, pp. 49–60, 2006.

Furthermore, a software can be tested on different levels and during different phases of development, which necessitates the use of different test design methods. Unit testing is usually associated with white box test design, because units are often of limited (structural) complexity. Testing of whole agent systems has to cope with increased complexity due to multitasking, nondeterminism and timing issues. Consequently, system testing focuses on the discovery of bugs that emerge from interactions of components or result from contextual conditions [4].

In the following we will focus on supporting testing of agents at the system's level.

2.1 Model-Based Testing of Agents

Whereas most of the testing approaches directly derive test cases from the specification of requirements, model-based testing uses additional models for test case selection. To constrain the set of test cases, model-based testing introduces a model of the test scenario to distinguish significant ones.

Types of abstraction that are used in model-based testing are: *functional, data, communicational*, and *temporal* abstraction [5]. Functional abstraction omits behavioral details that are of no interest according to the current test purpose, i.e. behavior is tested only with respect to a constrained environment. Data abstraction maps concrete data types to abstract ones. Communicational abstraction maps sets of interactions to atomic ones. Temporal abstraction moderates precise timing of events, e.g. to consider only the order in which events occur.

The complexity of an agent's environment makes functional abstraction mandatory for testing. Covering a certain set of test cases is hindered by the agent's autonomy. Because of its unpredictable behavior it is difficult to drive an agent into a certain test case [6]. Test cases should not be completely predefined but evolve dynamically depending on both the environment and the reactions of the agent itself [7]. In contrast to non-deterministic testing, environmental models provide a controlled selection of test cases.

3 Modeling Test Architectures

We suggest a rather radical interpretation of "model-based testing" by explicitly defining all abstractions in models. Therefore, we introduce a general model of the test architecture.

A test architecture must be easily adaptable to provide the required granularity and to complement the software under test as far as it has been developed. Thus, the chosen formalism and the test architecture should support modularity, re-use, and refinement. In particular, to be applicable for agent testing, the architecture should support multiple facets of testing and variable structures. Moreover, adequate testing of real-time constraints requires a formalism on a dense time base.

The approach presented here is based on James II, a component-based modeling and simulation system, which supports different modeling formalisms and facilitates different simulation engines [8]. Most of the realized modeling formalisms are based on DEVS [9] and extend the formalism by introducing variable structures [10] or peripheral ports into DEVS.

DEVS distinguishes between atomic and coupled models. An atomic model is described by a state set S, a set of input and output ports X and Y respectively, an internal and external transition function, δ_{int} resp. δ_{ext}, an output function λ, and a time advance function ta. δ_{int} dictates state transitions due to internal events, the time of which is determined by the ta function. At an internal event, the model produces an output in λ. δ_{ext} is triggered by external inputs that might arrive at any time.

Coupled DEVS models support the hierarchical, modular construction of models. A coupled model is described by the set of its component models, which may be atomic or coupled, and by the couplings that exist among them. Coupled models enable the structuring of large models into smaller ones.

3.1 Equipping DEVS with Peripheral Ports

To support the integration of externally running software, DEVS models have been equipped with peripheral ports (XP, YP).

The classical ports (X, Y) of DEVS models collect and offer events that are produced by models. In addition, the peripheral ports allow models to communicate with processes that are external to the simulation. Thereby, the simulation system does not interact with external agents as one black box, but each single model can function as an interface to external processes. The model functions were extended to handle input and output from respectively to external processes. The state transition functions and the $\lambda : S \times XP \rightarrow Y$-function of the interface model describe how incoming data is transformed into data that can be used within the simulation. The functions $\delta_{int} : S \times XP \rightarrow S \times YP$ and $\delta_{ext} : Q \times X \times XP \rightarrow S \times YP$ are furthermore responsible for the transformation of simulation data into data usable by the externally running software.

The simulation system uses the time model function $tm : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$ to translate the resource consumption of the externally running software into simulation time:

External processes are invoked by the simulation system and the information put into the peripheral output ports is forwarded to them. After the external computation has finished, the results of these invocations arrive at the peripheral input ports of the according model at a simulation time which is determined by the time model.

3.2 An Experimental Frame for Testing Agents

Models are designed, tested, and validated based on conditions and assumptions. The so called *experimental frame* [11] makes those explicit. Figure 1 depicts the components that make up an experimental frame for testing agents.

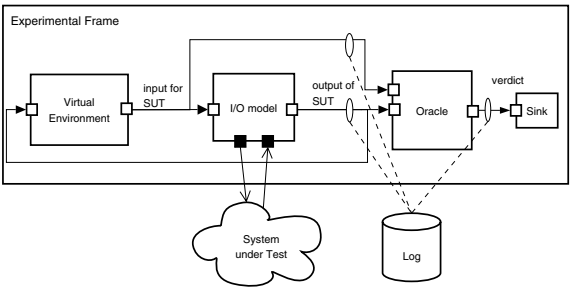

Fig. 1. Model of a test architecture for agents

I/O models form the interface between simulation and Software under test (SUT). It exchanges data between simulation and the externally running software via its peripheral ports. The state transition functions and the λ-function of the interface model describe how incoming data from the SUT is mapped to data that can be used within the simulation and vice versa. Thereby they describe the data abstraction in testing.

Moreover, *I/O* models also describe communicational and temporal abstraction. Using the time model $tm : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$ different types of temporal abstraction can be realized considering the progression of wall clock time and consumption of resources on the one hand and simulation time on the other hand. However, for testing timing requirements the validity of the chosen time model is crucial. Often the consumed wall clock time is used as a resource, as it is easily accessible. Unfortunately, its usage endangers the repeatability of simulation runs and implicitly introduces uncertainties due to hardware configuration and current work load. Better suited time models depend on the type of implementation, the language used, or the underlying operating system [12].

In all cases, an *ExternalProcessThread* has to be implemented per *I/O* model that realizes the actual communication between simulation and the SUT. If the implementation does not possess an own clock, a synchronous interaction of simulation and the external software will prove beneficial, as it gives the simulation system full control over the experiment.

The Virtual Environment models the circumstances under which the behavior of the SUT (i.e. the agent) is to be observed. Thus, it realizes the functional abstraction in testing and constrains the view on the possible behavior of the system. Within the virtual environment the peripheral ports can also be exploited for integrating user interactions, e.g. humans can interact asynchronously with the simulation supported by peripheral ports and a paced execution. Gradual transitions between explicit models and "ad-hoc" testing by humans become possible.

So far, we have merely formalized the experimental set-up for testing agents. Now, we shall see how this general architecture was deployed for testing a concrete agent application and how the general framework can be filled with application specific abstractions.

4 Testing Autominder

Autominder is a distributed monitoring and reminder system developed at the University of Michigan [13]. It is being designed to support older adults with mild to moderate cognitive impairment in their activities of daily living. These are activities a person must complete in order to ensure her physical well-being (e.g. eating, toileting). The software is installed in the client's house together with a set of sensors and given a list of actions that must be performed during the day. At runtime, Autominder evaluates sensor echoes from the environment, reasons about ongoing user activity, and compares its findings to the given client plan to detect forgotten actions and remind the user of their execution.

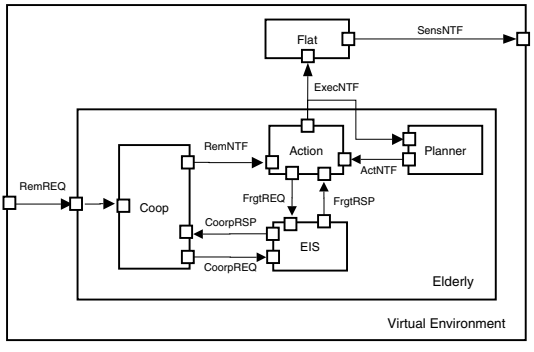

Fig. 2. Refined Virtual Environment for Testing Autominder

4.1 The Virtual Environment

Autominder complements a human in-place nurse and thus has a huge responsibility for its client's safety. Consequently, it must be tested in a variety of application scenarios before its release. Since those scenarios are not always safe for a human client and may be hard to observe during human-in-the-loop field tests, a virtual testing environment for Autominder (AVTE) has been developed [14] in cooperation with the Autominder research group. The AVTE was implemented in James II. Its layout is shown in Figure 2. The testing environment focuses on the model of an elderly who acts as the Autominder user. Besides, it includes a model of the client's living environment where Autominder is supposed to work.

The Elderly model is designed as a Human Behavior Representation that represents an elderly person in terms of actions and mental state [15]. Therefore, it provides two basic client functionalities: emulate suitable elderly behavior and react to incoming reminders. The virtual elderly follows a certain routine and performs actions of daily living accordingly. Depending on its initial memory fitness and current stress level, the Elderly model may forget some of the plan actions. If the model receives an external reminder, it interrupts this default behavior and evaluates the reminder. However, there is no guarantee that the

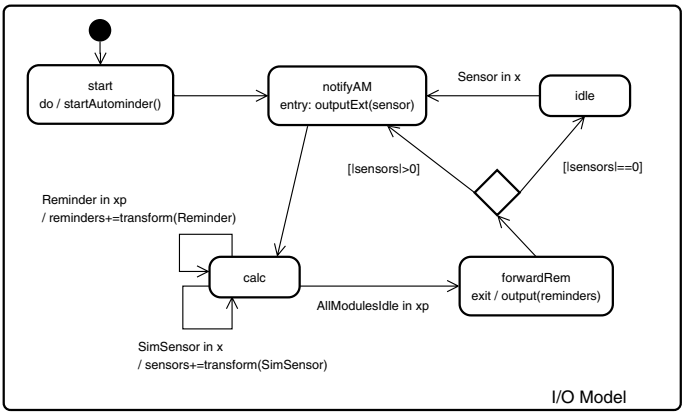

Fig. 3. An I/O model that functions as an ambassador for Autominder

Elderly model will cooperate with Autominder and execute the requested action. Depending on its cooperativeness and current degree of annoyance, the virtual elderly may choose to ignore reminders as well.

In the Elderly model, each of the basic functionalities is implemented by a separate component: the *Cooperation* model deals with reminder acceptance, and the *Action* model is responsible for the execution of actions. It receives information on currently executable actions from the *Planner* model that manages the routine plan of the elderly. The *ElderlyInformationStorage* (EIS) model holds all user-specific information necessary to compute the elderly's present forgetfulness and cooperativeness depending on the past workload.

The *Flat* model composes a sensory picture of the elderly's surroundings. It generates sensor data that reflect the current in-house activity based on the past action history and knowledge about how sensors are activated by plan steps. Examples for such sensors are contact sensors for doors and the medication container cover, heat sensors for the stove, and flush sensors in the bathroom.

4.2 Coupling Autominder and Simulation

For testing Autominder, the software has to be plugged into the virtual environment. Autominder is a complex and autonomous implementation. It offers an interface for external function invocation and is able to proactively start interactions itself.

To support both the testing of method invokation as well as the testing of agent software with an own thread of control, ambassador models have been introduced [16], which form a special kind of I/O models. With ambassador models a representative of the externally running software becomes part of the virtual environment, crucial phases of the software are directly reflected within the simulation model.

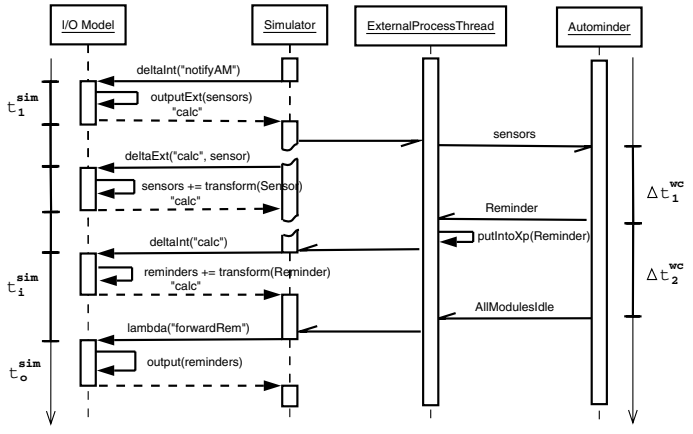

Fig. 4. Interaction between the ambassador I/O model and Autominder

Using ambassadors, the SUT typically needs to be instrumented, i.e. methods of the agent have to be changed to redirect messages into the simulation system. In the case of Autominder, its central module, the Event-Handler, was concerned. Every software module that wants to receive Autominder messages needs to be registered at the EventHandler, and so does the Ambassador model. Furthermore, some new message types were introduced. On the one hand, Autominder has to accept new simulation-generated synchronization messages in order to agree on a common time. On the other hand, marker messages must now be issued by the EventHandler so that the Ambassador model could clearly tell apart Autominder's different processing phases.

Figure 3 shows an ambassador *I/O* model for testing Autominder. This model reacts to incoming simulation messages by forwarding them to Autominder and reacts to incoming messages from Autominder by passing them to the other models of the virtual environment.

Communicational abstraction is used for collecting reminders and sensor notifications during phase *calc*. All reminders that have occurred during a computation cycle of Autominder are passed to the virtual environment jointly. Similarly, all sensor notifications issued by the virtual environment during the current Autominder processing cycle are gathered and held back until the transition to the phase *notifyAM* takes place.

Figure 4 shows a typical interaction between the ambassador *I/O* model and Autominder. Calculation phases of Autominder are initiated by the simulation system. Sensor information from the virtual environment is put into the peripheral output ports by the *I/O* model and forwarded to Autominder. Reminders produced by Autominder are put into the peripheral input port of the *I/O* model and saved until an *AllModuleIdle* message indicates that Autominder has finished a computation cycle. The simulation time of peripheral input events is determined by the time model, i.e. $t_i^{sim} = t_1^{sim} + tm(\Delta t_1^{wc})$, $t_o^{sim} = t_i^{sim} + tm(\Delta t_2^{wc})$. This is the time at which the reminders are scheduled in the virtual environment.

Autominder maintains an own clock, which is advanced either by Autominder itself or externally. Using ambassador models, Autominder time is controlled externally. Autominder advances its clock according to the time stamp of the simulation-generated messages it receives.

4.3 The Oracle

The oracle module judges, whether Autominder adequately reacts (issues reminders) with respect to formerly received inputs from the virtual environment. Adequacy is defined in terms of requirements that are specified at an earlier design phase. For testing Autominder's reminder generation feature, four requirements were identified in order to ensure client awareness and avoid annoyance respectively over-reliance:

1. All critical actions from the client plan shall be executed by the client. Therefore, Autominder needs to make sure the client is aware of upcoming activites: it has to issue reminders for actions the client does not perform on its own.
2. However, not all actions mentioned in the client plan may be essential and worth reminding for.
   Autominder must only remind for crucial client plan actions that are marked as "remindable".
3. If the client does not execute a such mandatory, remindable action on its own, it must be given the opportunity to catch up on time. Hence, reminders need to occur within the allowable time bounds of the related action.
4. In order to personalize reminder plans, both client and caregiver can give recommendations on when to remind best (e.g. earliest, latest, or typical execution time). Autominder's reminding policy should respect these preferences[1].

These requirements are translated into a set of rules the oracle uses in order to evaluate the correctness of Autominder's input/output trajectory.

4.4 Test Cases and Results

For each test case, input is specified by two plans: the routine plan that defines the behavior of the simulated elderly and the client plan based on which Autominder works. These plans can either agree or differ in their plan steps.

When using the same plans in Autominder and the elderly model, Autominder knows everything its user normally does during a day. Nevertheless, the elderly person may forget some of its usual activities. In this case, Autominder is expected to remind the client of the forgotten actions. In real-world applications, Autominder is likely to be given more sparse information on mandatory client activities only. Future testing scenarios should define the routine plan as a superset of the client plan steps, cf. Figure 5(a) and 5(b).

[1] In our test runs, not much emphasis was put on the adherence to the reminding preferences since "there is no good way to validate [...] [the] optimality [of the client plan]" [17].

action	start interval	end interval
getup	[1,1]	[2,2]
eat	[5,7]	[6,8]

(a) client plan

action	start interval	end interval
getup	[1,1]	[2,2]
wash	[3,3]	[4,4]
cook	[5,5]	[6,6]
eat	[7,7]	[8,8]

(b) superset routine plan

action	start interval	end interval
getup	[1,1]	[2,2]
wash	[3,3]	[4,4]
goOut	[5,5]	[6,6]

(c) differing routine plan

Fig. 5. Example relations of client and routine plan

In our first tests, we used the same plan as routine and client plan. A simple, strictly serial plan was employed. Figure 6 displays the six consecutive plan actions that start resp. end each planned activity and highlights the times when reminders were expected to occur for them. Flashes indicate the actual receipt of a reminder.

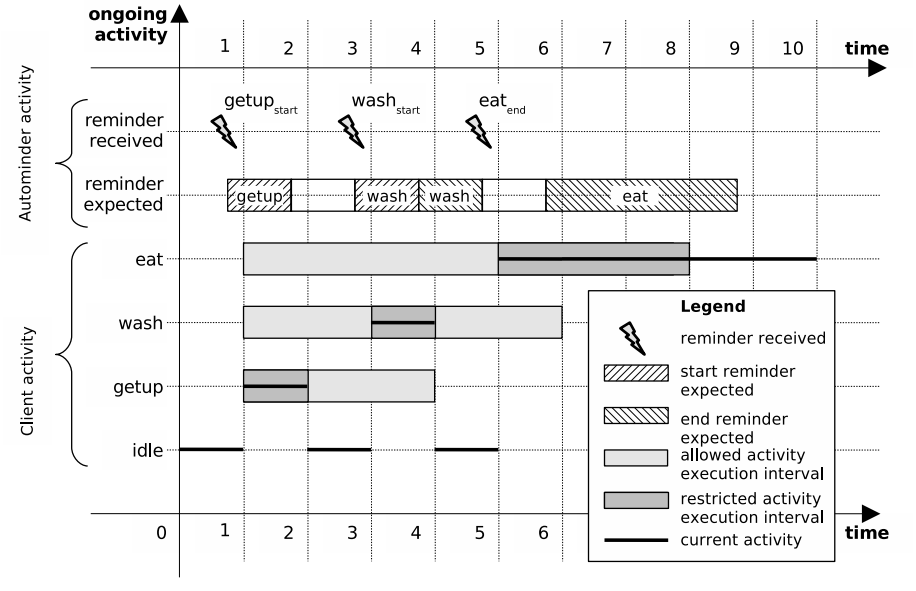

Fig. 6. Summary of simulation runs

In our experiments, reminders always occurred timely. So far, both requirement (1) and (3) are fulfilled. Interestingly, no end reminder was received during the simulation at all. GetUp and Wash both were ended by the model itself, no reminders were necessary. In contrast, the elderly model "forgot" to finish Eating. Here, the omitted end reminder caused problems – the elderly did not comply with its client plan anymore, requirement (1) is violated.

Moreover, this incident also represents an irregularity with requirement (2). Autominder shall remind for crucial actions only, but no reminder is expected for the start of eating. The evidence of a needless start and a missing end reminder nurtured the suspicion that there might be problems with end reminders in Autominder. However, in many safety critical situations ending an activity is an as crucial issue as starting an activity (e.g. turning off stove after cooking). Thus, more extensive testing of end reminders was conducted using an elderly model with a tendency to forget ending activities. The testing revealed that indeed Autominder did not consider end reminders yet. After having been identified, this problem could be resolved easily.

5 Related Work

There exist mature tools for testing real-time constrains by synchronously invoking implemented functions, e.g. TAXYS [18] and Simulink [19]. These works differ from ours in that the present approach also accounts for autonomous agent software.

Testbeds [20], [21] and [22] explored the integration of models rather than implementations of agents that are equipped with an own thread of control and communicate asynchronously with the simulation system.

TTCN-3 is a language for specifying test cases as well as test setups [23]. Whereas TTCN-3 aims to provide a standard notation and execution architecture for all black-box testing needs, we suggest a purpose-oriented modeling of tests and test architectures on an "appropriate" level of abstraction – according to the current development phase and the requirements of a software system.

The abstractions used in model-based testing were already discussed by Hahn *et al.* [6]. In our approach these abstractions became operational by using general purpose modeling and simulation methods.

James II has previously been used for coupling external software to simulation, e.g. synchronous coupling of external software to the simulation system and the use of time models were realized for planning agents [24], mobile agents, [16], and a multi-agent system comprising collaborative agents [12].

6 Conclusion

A modeling and simulation-based approach for testing agent implementations has been proposed. To this end, the idea of experimental frames has been adopted for defining test architectures.

The developed test architecture comprises different sub-models with different roles in the testing process. These sub-model reflect the different types of abstraction usually involved in testing.

Whereas the *Virtual Environment* model signs responsible for the functional abstraction, *I/O* models implement the data and communicational abstraction at least with respect to the communication between SUT and environment. The *Time Model* represents the temporal abstraction underlying the testing.

By using a formal modeling and simulation approach to testing, the mental models that form an intrinsic part of testing are explicitly represented. Furthermore, each of the models can be defined and refined on the level of abstraction required by testing purposes.

The practical use of the methods developed was demonstrated with the Autominder system. In this context particularly the importance of human behavior models became obvious. The integration of human behavior models is gaining importance in other areas, e.g. the evaluation of protocols for mobile ad-hoc networks [25], too. So this might be a general trend when it comes to simulation-based testing of software: to replace ad hoc interactions (of humans in-the-loop) with the software under test, by coherent (although not necessarily validated) and systematically responding user models.

Acknowledgments

This research is supported by the DFG (German Research Foundation).

References

1. Dam, K.H., Winikoff, M.: Comparing agent-oriented methodologies. In: Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems, Melbourne (2003)
2. Hilaire, V., Koukam, A., Gruer, P., Müller, J.P.: Formal specification and prototyping of multi-agent systems. In: ESAW 2000. Volume 1972 of Lecture Notes in Artificial Intelligence. Springer Verlag (2000) 114–127
3. Beizer, B.: Software Testing Techniques. 2nd edn. Van Nostrand Reinhold, New York (1990)
4. Bashir, I., Goel, A.L.: Testing Object-Oriented Software: Life Cycle Solutions. Springer (2000)
5. Prenninger, W., Pretschner, A.: Abstractions for model-based testing. In: Proc. Test and Analysis of Component-based Systems (TACoS'04), Barcelona (2004)
6. Hahn, G., Philipps, J., Pretschner, A., Stauner, T.: Prototype-based tests for hybrid reactive systems. In: Proc. 14th IEEE Intl. Workshop on Rapid System Prototyping (RSP'03), IEEE Computer Society (2003) 78–85
7. Kopetz, H.: Software engineering for real-time: a roadmap. In: ICSE - Future of SE Track, ACM Press (2000) 201–211
8. Himmelspach, J., Uhrmacher, A.M.: A component-based simulation layer for JAMES. In: Proc. of the 18th Workshop on Parallel and Distributed Simulation (PADS), May 16-19, 2004, Kufstein, Austria. (2004) 115–122
9. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. 2nd edn. Academic Press, London (2000)
10. Uhrmacher, A.M.: Dynamic Structures in Modeling and Simulation - a Reflective Approach. ACM Transactions on Modeling and Simulation **11**(2) (2001) 206–232
11. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press, London (1984)

12. Röhl, M., Uhrmacher, A.M.: Controlled experimentation with agents – models and implementations. In Gleizes, M.P., Omicini, A., Zambonelli, F., eds.: Post-Proc. of the 5th Workshop on Engineering Societies in the Agents World. Volume 3451 of Lecture Notes in Artificial Intelligence., Springer Verlag (2005) 292–304
13. Pollack, M.E., Brown, L., Colbry, D., McCarthy, C.E., Orosz, C., Peintner, B., Ramakrishnan, S., Tsamardinos, I.: Autominder: An Intelligent Cognitive Orthotic System for People with Memory Impairment. Robotics and Autonomous Systems **44** (2003) 273–282
14. Gierke, M.: Coupling Autominder and James. Master's thesis, University of Rostock (2004)
15. Gierke, M., Uhrmacher, A.M.: Modeling Elderly Behavior for Simulation-based Testing of Agent Software. Conceptual Modeling and Simulation CSM 2005 (2005)
16. Uhrmacher, A.M., Röhl, M., Kullick, B.: The role of reflection in simulating and testing agents: An exploration based on the simulation system james. Applied Artificial Intelligence **16**(9-10) (2002) 795–811
17. Rudary, M., Singh, S., Pollack, M.E.: Adaptive Cognitive Orthotics: Combining Reinforcement Learning and Constraint-Based Temporal Reasoning. 21st International Conference on Machine Learning (2004)
18. Sifakis, J., Tripakis, S., Yovine, S.: Building models of real-time systems from application software. Proceedings of the IEEE **91**(1) (2003) 100–111
19. MathWorks: Simulink. `http://www.mathworks.com/products/simulink/` (2005)
20. Pollack, M.E.: Planning in dynamic environments: The DIPART system. In Tate, A., ed.: Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative, AAAI Press, Menlo Park, CA (1996) 218–225
21. Anderson, S.D.: Simulation of multiple time-pressured agents. In: Proc. of the Wintersimulation Conference, WSC'97, Atlanta (1997)
22. Kitano, H., Tadokoro, S.: RoboCup Rescue: A grand challenge for multiagent and intelligent systems. AI Magazine **22**(1) (2001) 39–52
23. Wiles, A.: ETSI testing activities and the use of TTCN-3. Lecture Notes in Computer Science **2078** (2001) 123–128
24. Schattenberg, B., Uhrmacher, A.M.: Planning agents in James. Proceedings of the IEEE **89**(2) (2001) 158–173
25. DIANE-Projekt: Dienste in Ad-Hoc-Netzen. `http://www.ipd.uni-karlsruhe.de/DIANE` (2005)