UNPACED AND PACED SIMULATION FOR TESTING AGENTS

Adelinde M. Uhrmacher
Mathias Röhl
Jan Himmelspach
Universität Rostock, Department of Computer Science
Albert-Einstein-Str. 21, D-18059 Rostock, Germany

KEYWORDS

Parallel, distributed Simulation, Multi-Agent Systems, Paced and Unpaced Simulation

ABSTRACT

Agents are autonomous software aimed at working in dynamic environments and thus form a specific type of embedded software systems. To test this type of software simulation systems can be successfully employed. Agents might be modeled, be partly embedded in, or coupled to the virtual environment they are tested in. Depending on the degree of being embedded in the virtual environment, the type of execution that supports an efficient and effective simulation varies. In JAMES (A Java-Based Agent Modeling Environment for Simulation) different simulators have been implemented. Unpaced and paced simulators support interaction in simulation- and real-time differently. Moving from unpaced to paced execution, the simulator exercises less control over the experiment and the coupling between simulation and agents to be tested is loosened.

1 INTRODUCTION

Agents can be interpreted as software systems that are aimed at working autonomously in dynamic and uncertain environments (Jennings et al. 1998). The interrelations between agents and simulation are manifold (Uhrmacher et al. 2001). Software agents are used to develop "state of the art" simulation systems and agents are used as a metaphor for modeling dynamic systems as collections of autonomously interacting entities. Software agents are often mission critical (or even safety critical) and, like other software systems, must be tested and evaluated before being deployed. Their autonomy and the open heterogeneous nature of the environment in which they operate make testing and evaluation more difficult than in the case of more conventional software systems. As agents are aimed at working in dynamic environments, simulation seems a natural approach towards testing the behavior of an agent system in interaction with its environment.

The implementation and application of dynamic test scenarios for multi-agent systems require considerable modeling effort. Already early simulation systems for agents allowed to plug code fragments, or single modules into the skeleton of an agent model (Montgomery and Durfee 1990). Others treat agents as external source and drain of events (Pollack 1996). The continuity of models from specification via simulation to implementation shall help reducing flaws during the design of software systems (Hu and Zeigler 2002). However, it is less the continuity of models during designing agent systems we will discuss in this paper but the different types of simulators that accompany the different stages of developing agents.

Based on JAMES (Schattenberg and Uhrmacher 2001) (A Java based Agent Modeling Environment for Simulation) and based on the project AUTOMINDER (Pollack et al. 2003) that is aimed at developing a planning agent software supporting elderlies in their homes, we will illustrate our approach for testing agents.

2 JAMES

JAMES has been developed based on the formalism DYNDEVS. The model design of JAMES resembles that of DEVS (Discrete Event System Specification) (Zeigler et al. 2000) extended by means for reflection which allows agents to adapt their composition, interaction, and behavior patterns. Models can create new models and add them to the coupled model they belong to, they can remove themselves, and they can access their interaction structure. To initiate structural changes outside their

Proceedings 15th European Simulation Symposium
Alexander Verbraeck, Vlatka Hlupic (Eds.)

ISBN 3-936150-28-1 (book) / 3-936150-29-X (CD)

boundary, agents have to turn to communication and negotiation. Thus, a movement from one coupled model to another implies that another atomic model complies with the request to add the moving model into the new interaction context. To facilitate modeling, all atomic models are equipped with default methods that allow them to react to those requests. However, these default reactions can be suppressed to decide deliberately what requests shall be executed. The freedom to decide whether to follow a certain request, e.g. to commit suicide, and its knowledge, i.e. beliefs about itself and its environment, distinguish active agents from more "reactive" entities (Jennings et al. 1998). The ports of DEVS models, which are used for communication between models, are complemented by peripheral ports in JAMES. Models communicate via peripheral ports with processes that are external to the simulation. Time models allow to transform the resource consumption of the external processes into simulation time (Schattenberg and Uhrmacher 2001; Riley and Riley pear). The modular, hierarchical modeling concept facilitates the re-use of components and thus the construction of virtual test environments by composition.

JAMES has been used for testing different types of software agents. As other simulation systems (Montgomery and Durfee 1990) it allows to plug in code fragments, or single modules, whereas the agent itself is specified as part of the model. We followed this approach by testing planning agents in a TILEWORLD scenario in JAMES (Schattenberg and Uhrmacher 2001). In later phases, the ability to execute agents as they are, and to switch arbitrarily between an execution in the real environment and the virtual test environment gains importance. If simulator and agent software are only loosely coupled (Pollack 1996; Anderson 1997), agents are typically only perceivable by their effects in the virtual environment and no longer really controlled by the simulation. To bridge the gap between earlier and later phases in designing agents and to support the continuity of models, the idea of representatives has been introduced in JAMES. The idea took concrete form in testing and plugging agents of the mobile agent system MOLE into JAMES (Uhrmacher et al. 2002).

3 THE EXAMPLE - AUTOMINDER

AUTOMINDER is a software agent designed as a cognitive orthotic which shall assist elderlies with memory impairment in carrying out their daily life activities. Therefore, the activities of elderlies are monitored and elderlies, if they forget or confuse certain activities, shall be reminded in a timely and adequate manner (Pollack et al. 2003; McCarthy and Pollack 2002; et al 2002). The software is being developed in the context of the Initiative on Personal Robotic Assistants for the Elderly (Montemerlo et al. 2002) and installed on the nursebot PEARL.

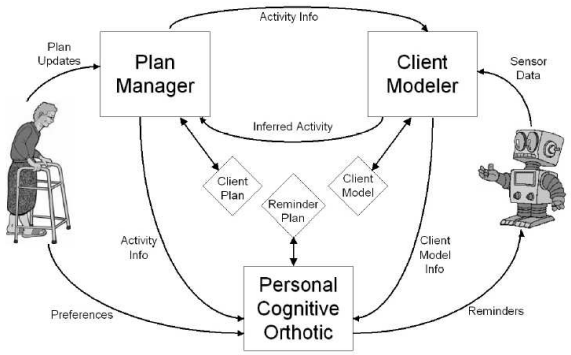

Figure 1: Structure of the AUTOMINDER Agent (Pollack et al. 2003)

The architecture of AUTOMINDER comprises the components: Client Modeler (CM), Plan Manger (PM) and Personal Cognitive Orthotic (PCO). The role of the CM is to interpret the sensor information and based on a given plan to identify activities that are just started or being ended and to notify the Plan Manager. The CM maintains and updates a client model which contains observations and is used to derive regular behavior patterns of the elderly. The Plan Manager maintains a plan in terms of activities the elderly is supposed to perform, i.e. the client plan. Subsequently the plan can be updated by the caregiver and to a certain degree also by the elderly. The plan manager checks the client plan for inconsistencies and is responsible for resolving potential conflicts. Based on the client plan and the client model, the PCO finally decides what kind of reminders to launch at which time.

The AUTOMINDER agent combines reactive and deliberative abilities, like planning and learning. Taking opportunities into account while not loosing track of pursued goals is seen as one of the central challenges in developing successful agent software (Wooldridge and Ciancarini 2000). Therefore, the mediation between reactivity and deliberation and its effect on the performance of agents has traditionally been at the core of evaluating hybrid agents in small play world scenarios e.g. (Cohen et al. 1989; Kinny et al. 1996; Schattenberg and Uhrmacher 2001).

4 TOWARDS A MODEL-BASED TESTING OF AUTOMINDER

Some time has elapsed since Paul Cohen, Steve Hanks, and Martha Pollack wrote their paper on controlled experimentation, agent design, and associated problems (Hanks et al. 1993). Their controversy about testing in the small and testing in the large in designing agent systems has neither lost its topicality nor its virtue, though. Test beds, e.g. DVMT (Durfee 1988), PHOENIX (Greenberg and Westbrook 1990), TILEWORLD (Pollack and Ringuette 1990), soccer game (Kitano et al. 1997), and large scale disasters (Kitano et al. 1999) represent a complement to conventional benchmark tests, offering test scenarios which are aimed at revealing prototypical problems in dynamic environments. Within this testing in the small, it is not the purpose to confront the agent with a valid model of the concrete environment the agent shall dwell in. In contrast "testing in the large" is based on test cases that shall emulate requirements of the real environment. Often test cases are based on and sometimes even automatically generated from software requirements, source-code statements, and module interfaces (e.g. (Peraire et al. 1998)). However, whereas for many embedded systems a clear specification of the software and the required functionality exists, this is not necessarily true for agent systems. Instead Wooldridge and Jennings observe that "the development of any agent system - however trivial is essentially a process of experimentation" (Jennings and Wooldridge 1998). Therefore, experimentation has been part of developing agents from the very first.

AUTOMINDER's activities are constrained by time: it has to react timely and appropriate to the elderlies actions and reactions. Most agent's decisions and deliberations are limited by time. The timeliness and adequacy of their reaction determine their performance. AUTOMINDER's activities are triggered by the flow of time: many of the activities of elderlies are scheduled for certain times of the day and the robot has to remind the elderly in time if these activities are crucial for the elderlie's health. So AUTOMINDER displays situation-triggered and time-triggered activities.

If AUTOMINDER is tested in its real environment, interaction happens in physical time. A significant evaluation of the performance of AUTOMINDER would likely take at least a month — besides the problem to find suitable test persons for the experiments. The functionality of AUTOMINDER can not be tested based on one time point only. Its interaction with the environment has to be observed over a period of time. Thus, simulation seems a natural approach towards testing the behavior of this agent system in interaction with its environment.

Environment models are used to generate the different test cases dynamically during simulation, including specific interaction patterns and time constraints (Schütz 1993, p.23). The focus of model-based testing shifts from the specification of the software to modeling the dynamic environment of the agent.

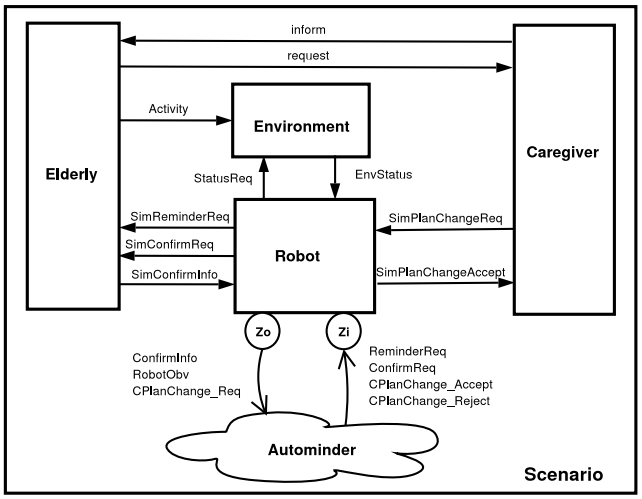

Figure 2: Structure of the JAMES Model for Testing the AUTOMINDER Software

Using simulation for testing software means to develop software, including models, routines for initialization of models and the simulation engine. Particularly, if software is used for testing other software, it is crucial that certain quality characteristics of the software product, including accuracy, can be assured. Component-based development of software is valued as an effective and affordable way to facilitate verification, validation and accreditation of modeling and simulation applications (Balci et al. 2002).

To ease the process of obtaining confidence that the model can be considered valid for its intended application (Sargent 1999), a component-based design of the model, where model components can be inspected separately, and the simulation system, which supports different approaches towards testing (see next section), has been chosen. The virtual environment in which AUTOMINDER shall be tested is built up of four different model components (Fig. 2) that can be evaluated separately. The model component `Elderly` represents the client to be supported by AUTOMINDER. The `Robot` represents a nursebot endowed with AUTOMINDER. Coupling AUTOMINDER, which runs concurrently and externally to the simulation, and the virtual environment is done by utilizing the `Robot` component as an interface between simulation and agent software. AUTOMINDER sends its time labeled events to the robot who charges its output ports with these events at the specified time. From the ports they are automatically transferred to the other models as defined by the couplings. The robot has to

explicitly request new status information about its environment from the model `Environment`. The model `Caregiver` forwards new plans to the `Robot` and AUTOMINDER.

The intention of our experiments with AUTOMINDER is behavioral testing or black box testing. The goal of AUTOMINDER is to provide elderlies with timely and appropriate reminders. This implies that AUTOMINDER has to find a balance between maximizing the elderlies compliance in performing his or her daily activities, maximizing the satisfaction of elderly and caregiver with the system, and avoiding making the elderly overly reliant on the system (Pollack et al. 2002). To achieve these goals the system has to be adaptable to different types of elderlies, actors, and circumstances which have to be represented with a sufficient accuracy.

So far AUTOMINDER has been tested manually: a user informs interactively AUTOMINDER about activities of the elderly and advances the virtual time. The goal of experiments based on JAMES is to test the behavior and the adaptation strategies of AUTOMINDER by using different model components representing explicitly different types of elderlies and domestic environments.

5 SIMULATOR

Executing the model according to the user's specification and the given initial situation is the task of a discrete event simulator. Simulation models are interpreted and executed by a tree of processors, which reflect the hierarchical compositional structure of the model (Fig. 3). Each of the processors is associated with a component of the model and is responsible for invoking the component's methods and controlling the synchronization by exchanging messages with the other processors of the processor hierarchy. The change of model structure is reflected in an according change of the processor tree. Different distributed, parallel execution strategies have already been implemented in JAMES based on the abstract simulator introduced in DYNDEVS (Uhrmacher and Gugler 2000; Uhrmacher and Krahmer 2001). Whereas one adopts a conservative strategy where only events which occur at exactly the same simulation time (including starting external processes) are processed concurrently, two other strategies split simulation and external processes into different threads and allow simulation and deliberation to proceed concurrently by utilizing simulation events as synchronization points. All simulators currently execute in an unpaced mode which means that simulation time does not elapse in relation to wall clock time but jumps as fast as possible from one event to the next, neglecting the simulation time (and thus the represented physical time) that lies

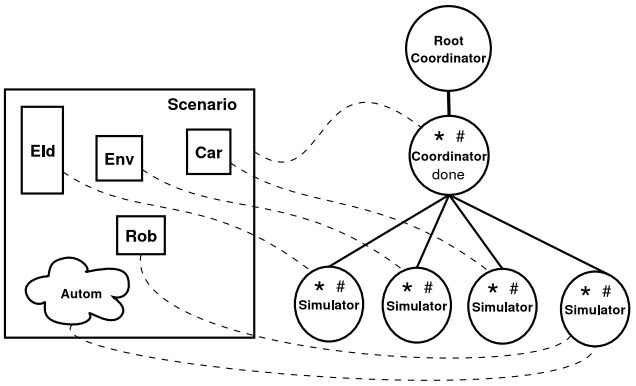

Figure 3: Models and Corresponding Simulator Tree in JAMES

inbetween (Fujimoto 2000). Within the limits that are determined by the wall clock time required to execute events, paced simulations can be scaled to allow a faster or a slower progression of time. The scaling factor can be changed during simulation, to skip in a fast mode through less interesting episodes and to zoom in to explore interesting episodes in detail. Paced simulations are typically used for training humans, whereas unpaced simulations are used for analytical purposes. Both allow the interaction with humans and with external soft- or hardware. However, their means and also their focus is different.

5.1 UNPACED SIMULATOR

The following conservative simulator supports an as-fast-as-possible discrete-event-simulation and exploits the parallelism inherent in concurrently deliberating multiple agents.

Figure 4 describes the *-handler of the simulator, which is at the core of the overall simulation algorithm. Besides a *-handler, simulators are equipped with a #-handler for dealing with inputs. Coordinators have additionally a `done`-handler which records that events have been processed or guarantees are given.

The simulator of a model is activated by the *-message, which indicates an internal, external, or confluent event. With the label *guarantee*? set to true the *-handler of the simulator is asked to guarantee that none of its pending external processes will finish before the time t.

The set *busy* is updated by the external agent processes which adds $busy_i$ to it in the moment the process i is started. If the process has finished, $busy_i$ is deleted from the set *busy*. The resource consumption of the process is recorded, as are the results with which the

```
when an input (∗, guarantee?, xCount, t) has been received
  am is the associated model
  outCount = 1
  if guarantee?  then
    block until for all i ∈ busy_fixed
      t_start_i + timeModel(i) > t ∨ i ∉ busy
    justFinished = {i|i ∈ busy_fixed \ busy}
    for all i ∈ JustFinished
      t_finished_i = t_start_i + timeModel(i)
      t_finished = t_finished ∪ {t_finished_i}
      t_start = t_start \ {t_start_i}
      busy_fixed = busy_fixed \ {busy_i}
    send (done, min(t_next, t_finished), ∅,
        outCount, (busy_fixed ≠ ∅)) to parent coordinator
  else
    if ¬ guarantee?  then
      inpCount = xCount
      t_min = min(t_finished, t_next)
      t_finished = t_finished \ {t_finished_i ∈ t_finished|t_finished_i = t}
      if t = t_min then
        if t = min(t_finished) then flush z_i
        send (λ(s, z_i)) to parent
        if xCount = 0 then
          (s, z_o) = δ_int(s, z_i)
        else
          block until inpCount = 0
          (s, z_o) = δ_con(s, xb, z_i)
        endif
      else
        block until inpCount = 0
        (s, z_o) = δ_ext(s, t − t_last, xb, z_i)
      endif
      for all i ∈ busy \ busy_fixed
        t_start_i = t
        t_start = t_start ∪ {t_start_i}
        busy_fixed = busy_fixed ∪ i
      am = ρ(s)
      t_last = t
      t_next = t_last + ta(z, s)
      send (done, min(t_next, t_finished),
          varStrucRequest(s), outCount, (busy_fixed ≠ ∅))
          to parent coordinator
    endif
  endif
end
```

Figure 4: The *-handler of the unpaced simulator in JAMES

peripheral input ports z_i shall be charged. The procedure which starts the external process within a separate thread generates unique names for the processes with which its start time, the finish time, and the results are labeled. $busy_{fixed}$ contains the processes that the simulator believes to be running. *busy* contains the processes which are actually running. t_{start} embraces the starting time of all processes the simulator believes to be running. It is incremented when a new process is started and decremented when the simulator discovers a completion. $t_{finished}$ lists the completion time of all processes of whose completion the simulator is aware.

If at least one external process is running the simulator blocks until each of the processes running has

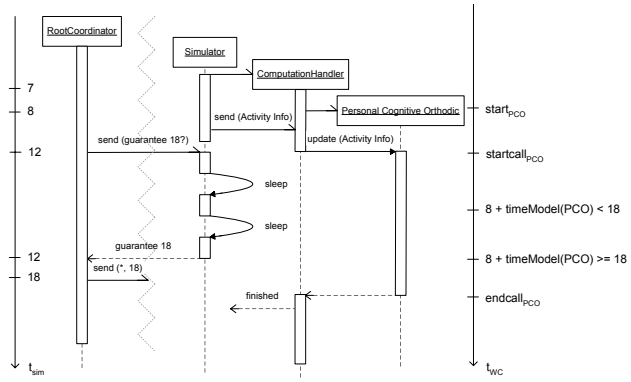

Figure 5: The Interaction between Simulation and AUTOMINDER in the Unpaced Version in JAMES

either reached the current simulation time or has been finished. The processes which just have been finished are determined, and for all of them the virtual time of finalization is calculated. The sets $t_{finished}, t_{start}$ and $busy_{fixed}$ are updated.

If *guarantee*? is set to false, the *-message announces an event and a confluential δ_{con}, internal δ_{int}, or external δ_{ext} transition is due. If at the current time the completion of a deliberation process is scheduled, the peripheral port z_i is charged with the results of this deliberation process. The transition functions update the state and generate outputs which are directed towards externally running software, e.g. invoking a deliberation process. After executing the transition function it is checked whether new processes have been started. Their starting time is determined and the sets t_{start} and $busy_{fixed}$ are updated. Afterwards structural changes are executed at the level of the atomic model by invoking the model transition ρ. The time of last and time of regular next event are determined. The coordinator is informed of the time of next event and whether any structural changes at the level of the coupled model are due. If agents deliberate sufficiently long and a sufficient number of processors are available, multiple deliberating agents can be executed nearly at the cost of one deliberating agent (Uhrmacher and Krahmer 2001).

In the unpaced simulation (Fig. 5), the simulator controls the execution of agents, by invoking the methods, recording the simulation time when the external software has been started and by transforming the resources consumed during the execution of external software into simulation time to determine when the execution has been finished in simulation time. The simulation, its view of the world and particularly its view on time, controls the experiment: the software is executed embedded in the virtual environment. Performance criteria can be calculated, and bottlenecks can be identified. The user can interactively set parameters by exe-

cuting the simulation in stepping mode or by introducing break-points in the simulation. This version of the simulator is suitable if separate modules of AUTOMINDER shall be tested, e.g. the personal cognitive orthotic and specific methods of AUTOMINDER are invoked. The orthotic module of AUTOMINDER receives information about the elderly including his or her whereabouts and the originally planned schedule of the elderly. This information is provided by the client and plan manager module of AUTOMINDER. Part of its functionalities are now part of the modeled robot. In latter stages of designing AUTOMINDER the control of the simulator might be slightly loosened.

5.2 PACED SIMULATOR

The following simulator is a first attempt to support a paced simulation and an asynchronous exchange of messages between simulator and external software. In paced simulation: each advance in simulation time is paced to occur in synchrony with a scaling factor times an equivalent advance in wall clock time.

```
when an input (*, xCount, t) has been received
  am is the associated model
  outCount = 1
  inpCount = xCount
  block until W2S(WallclockTime)1) >= t
  flush z_i
  if t = t_next then
    send (λ(s, z_i)) to parent
    if xCount = 0 then
      (s, z_o) = δ_int(s, z_i)
    else
      block until inpCount = 0
      (s, z_o) = δ_con(s, xb, z_i)
    endif
  else
    block until inpCount = 0
    (s, z_o) = δ_ext(s, t − t_last, xb, z_i)
  endif
  am = ρ(s, z_i)
  t_last = t
  t_next = t_last + ta(s, z_i)
  send (done, t_next, varStrucRequest(s), outCount) to
    parent coordinator
end

1) W2S(t_wc) = t_simStart + scale ∗ (t_w − t_wcStart)
```

Figure 6: The *-Handler of a Simulator in JAMES - Paced

Each simulator blocks until its local virtual time has reached the wall clock time (Fig. 6). The simulator is a scaled paced simulator which allows to let the simulation run twice or half as fast as wall clock time. The speed up of simulation is constrained by the execution speed of simulation events and by the execution speed of the external software. Every time an event takes place

```
t_next = t_next(topmost coordinator)

repeat until t_next > t_EndOfSimulation ∨ (t_next = ∞)
  while W2S(wallclockTime) < t_next − ε
    if externalMessageArrived
      externalMessageArrived = False
      t_next = W2S(wallclockTime)
    endif
  endwhile
  send (*, 0, t_next) to topmost coordinator
  wait for (done,t,varStruc,outCount)
  from topmost coordinator
  t_next := t
end
```

Figure 7: The Root Coordinator in JAMES

the peripheral ports are flushed, i.e. they are read and emptied afterwards. Simulator and external software exchange messages in an asynchronous manner.

The simulation is notified that messages from external software systems have been arrived and thus have to be processed. The notification is propagated up the tree towards the root coordinator. For propagating the notification, the done threads of the coordinators are used which mark the coordinators along the way and thus allow afterwards to trigger top down the correct components of the processor tree. During their propagation upward, notifications are caught by *-messages traveling down the processor tree. In this case the current simulation pulse will be used for processing. Only if a message arrives in a sufficiently large gap inbetween events scheduled in the simulation, it will reach the root coordinator and trigger a simulation pulse.

The root coordinator is traditionally responsible for advancing simulation time in JAMES. Also in the paced variant it controls the advance in simulation time. Therefore, it blocks the simulation for some time before informing its children. If this were not the case and only the simulators were responsible for blocking the simulation, an external software could not trigger any event before the next scheduled event at t_{next} would have been executed. Holding the simulation at the root coordinator allows that the external software can trigger events between the current simulation time and the time of scheduled events. The ϵ accounts for real time delays that are caused by propagating messages from the root to the simulators. The idea is to choose ϵ sufficiently large, so that the simulators are triggered by the *-message in advance to be able to block $W2S(WallclockTime)^{1)} >= t$ and do not fall behind the wall clock time. However, this blocking time of the simulators should be minimized. Since during the time the simulators (and coordinators) are processing events, the root coordinator is waiting for done messages and will not acknowledge incoming messages from external

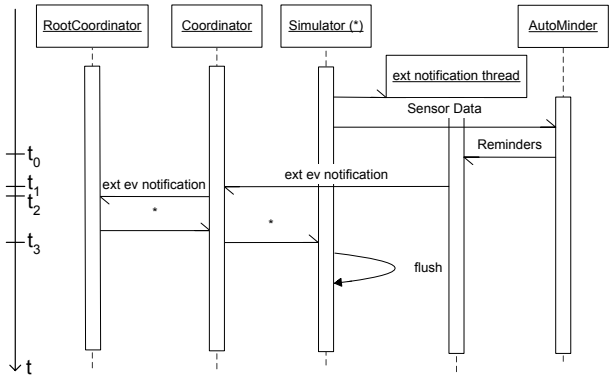

Figure 8: The Interaction between Simulation and AUTOMINDER in the Paced Version in JAMES

software systems. Messages will be handled promptly by the simulator however with a possible delay which is caused by propagating messages up and down the processor tree, and processing current events (Fig. 8).

Time stepped execution is the predominant model for paced simulation. Particularly in combination with visualization the procedure seems appropriate. The simulation is executed as a series of discrete intervals of time Δt. In each time interval messages are retrieved, the new internal state is computed, and messages are sent to other logical processes. After this the internal time is incremented by Δt and suspended until the time is reached. All messages are delivered in receive order and assumed to be relevant for now. Zeigler and his students adapted this approach and implemented a real-time event-based simulator (Cho et al. 2000). As the simulator exchange messages in receive order and not in time stamp order, no coordinator or root coordinator is necessary. Compared to this procedure of real-time execution our paced execution realizes a kind of hybrid. The proposed paced simulator asynchronously communicates with external software, both, simulator and agents, exchange information in receive order. However, within the simulation events are executed in strict time stamp order and resemble the typical analytic simulation. By supporting time stamped events in the distributed simulation of JAMES an overhead is induced. E.g. currently events that are initiated by messages from external software are processed at a simulation time that lags wall clock time. Even though first experiments have shown that the delay seems not critical, a more systematic analysis of the behavior of the simulator is necessary. For this purpose yet another real-time simulator is currently being implemented that will process events in receive order and not in time stamp order. In both simulators the problem of ensuring repeatability of simulation needs still to be addressed (McLean and Fujimoto 2000).

The role of the robot model is currently only to mediate between the dynamic virtual environment and AUTOMINDER: it frequently requests status information about the environment, information from the environment are forwarded to AUTOMINDER, and the output of AUTOMINDER is redirected into the simulation model and forwarded to the elderly. In other experiments, e.g. when the information about the elderly is transmitted directly by sensors in the flat, more detailed models about the soft- and hardware environment of the AUTOMINDER system will be probably required.

The paced version is used to loosely couple the entire AUTOMINDER software to JAMES. To tune the simulation to run faster than real time, the internal clock of AUTOMINDER has to pace time with the same scale as the simulator does.

6 DISCUSSION

Testing activities support quality assurance by gathering information about the nature of the software being studied. Little work has been done so far on developing methods for testing agents (Dam and Winikoff 2003). Whereas verification deals with transformational accuracy, validation deals with behavioral or representational accuracy. The analysis of temporal properties of software can be done statically or dynamically, the later is based on the execution of software.

Same as the functionality of real-time systems and embedded systems, the functionality of agent systems can not be evaluated based on an a priori fixed input specifications, but only as a course of reactions to the evolution of an environment. As agents are aimed at working in dynamic environments, simulation seems a natural approach towards testing the behavior of an agent system in interaction with its environment. Its interaction with the environment has to be observed over a period of time. The usage of a virtual environment in contrast to the real environment typically reduces costs and efforts and allows to test system behavior in "rare event situations". Virtual environments are easier to observe and to control, and probe effects are easier to manage. Simulation is mainly seen as a tool to validate temporal properties of *models* (Edwards et al. 1997). We like to widen the usage of simulation to embrace validation of the final software product as well, i.e. to use simulation throughout the development process. Execution monitoring, profiling and tracing, as provided by the simulation system, can be employed to assess the performance of the agent software. The simulated environment facilitates the testing of agent's behavior when confronted with extreme inputs and put under real-time pressure.

The simulation application itself is a software product and as such must be tested, i.e. verified and validated, in order to install confidence into the executed tests. Model verification implies that model transformation occurs with a sufficient accuracy. To verify the simulation it is required to analyze whether correct trajectories according to the model description are produced. The model verification does not really apply to our case. However, the simulation verification is important to assure that the simulation is not only efficient but also produces reliable results. Ongoing work is dedicated to analyze the repeatability of simulation runs and the role of the synchronous and asynchronous interaction of simulation and external software in more detail. The design of simulators as components and based on components requires a careful analyzing of the components, in isolation and once integrated, and to determine the context for which the developed components are truly exchangeable (Weyuker 1998).

To develop valid models, e.g. of the surroundings the agent AUTOMINDER is supposed to dwell in, poses serious difficulties. To validate the elderly model component data about the interaction of elderlies and nursebots are required. As the testing of basic functionalities of the nursebot robot in interaction with elderlies has just started (Montemerlo et al. 2002), a true validation of the elderly models in daily life seems out of reach. The best we can do is developing a set of prototypical and plausible model components that mimic the behavior of elderlies and a couple of bizarre ones to test the behavior of AUTOMINDER in borderline cases. Based on these models the simulation generates test cases dynamically, taking the activities of AUTOMINDER into account. Given the complexity of possible environment and AUTOMINDER interactions an exhaustive testing is not possible. To trace the execution of AUTOMINDER under a wider range of circumstances, AUTOMINDER can be tested in a virtual world whose model components include stochastic aspects. For this type of execution monitoring over many simulation runs the unpaced simulation will likely prove more suitable and the most interesting traces might be replayed by using the paced simulator. Thus, the temporal development of boundary cases can be analyzed in detail based on the paced simulator, whereas the unpaced simulation helps to identify these cases. Different simulators provide different approaches towards testing software, which eases the general experimentation, helps to reveal flaws within the tested system as well as in the used environmental models and simulators, thereby increasing the confidence into the used models and testing via simulation in particular.

As with all testing it is important that different groups are developing the software and the test scenarios. Independent verification and validation is a technique of long standing in the field of software engineering (IEEE 1998, p.58). The simulation software, particularly its usability, is evaluated by the AUTOMINDER research group. For this assessment, besides the offered functionalities, the user interface which is currently under development will play a crucial role. A close interaction with AUTOMINDER research group is currently directing the development of our models. However, as soon as the models have left the primordial ooze another group of the Nursebot project which is working closely with the elderlies shall help to evaluate the developed models. The more the project progresses the more the different research groups will be able to serve as a kind of software quality assurance group for each others software which answers part of the question "quis custodiet ipsos custodes" (who is guarding the guards).

7 CONCLUSION

The unpaced conservative simulator and the paced simulator are aimed at testing multi-agent systems containing a small number of deliberative, resource intensive agents. However, both simulators offer a different degree of control. The unpaced simulator invokes methods of the external software, responses of which are fed back into the simulation in simulation time. Our experiences with earlier agent projects indicated that as agents move from specification to implementation, different types of simulators are required. In the current project of coupling JAMES and AUTOMINDER we will explore this relation in detail. In this context a paced simulator is being developed, which advances simulation time in synchrony with wall clock time and supports a more realistic view on the virtual world. Agents are only loosely coupled to the simulator which facilitates a plug and play for the agent programmer. Whereas for earlier stages of developing agents an unpaced simulator seems appropriate for a first test of design decisions, later stages of implementation benefit from the easy plug and play and the more realistic view on the simulated environment the paced simulator provides in combination with an asynchronous exchange of messages. However, we expect that at some point in the designing process of agents performance issues of the complete software shall be analyzed again, and thus the unpaced simulator might come in handy.

ACKNOWLEDGMENTS

This research is supported by the DFG (German Research Foundation). We would also like to thank the anonymous reviewer who motivated us to include the discussion section.

REFERENCES

Anderson, S. (1997). Simulation of Multiple Time-Pressured Agents. In *Proc. of the Wintersimulation Conference, WSC'97*, Atlanta.

Balci, O., R. E. Nance, J. D. Arthur, and W. F. Ormsby (2002). Expanding our horizons in vv&a research and practice. In *Proceedings of the 2002 Winter Simulation Conference (San Diego, CA, Dec. 8-11)*, pp. 653–663. IEEE, Piscataway, NJ.

Cho, Y., B. Zeigler, H. J. Cho, H. S. Sarjoughian, and S. Sen (2000). Design considerations for distributed real-time DEVS. In *Proceedings of Artificial Intelligence and Simulation*.

Cohen, P. R., M. L. Greenberg, D. M. Hart, and A. E. Howe (1989). Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. *AI Magazine 10*(3), 32–48.

Dam, K. H. and M. Winikoff (2003). Comparing agent-oriented methodologies. In *Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems*, Melbourne. to appear.

Durfee, E. H. (1988). *Coordination of Distributed Problem Solvers*. Boston: Kluwer Academic Publishers.

Edwards, S., L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli (1997). Design of embedded systems: Formal models, validation, and synthesis. *Proc. of the IEEE 85*(3).

et al, M. P. (2002). Pearl: Mobile robotic assistant for the elderly. In *AAAI Workshop on Automation as Eldercare, August 2002*.

Fujimoto, R. (2000). *Parallel and Distributed Simulation Systems*. John Wiley and Sons.

Greenberg, M. and D. Westbrook (1990). The phoenix testbed. Technical Report UM-CS-1990-019, Computer and Information Science, University of Massachusetts at Amherst.

Hanks, S., M. E. Pollack, and P. R. Cohen (1993). Benchmarks, Test Beds, Controlled Experimentation and the Design of Agent Architectures. *AAAI* (Winter), 17–42.

Hu, X. and B. Zeigler (2002). An integrated modeling and simulation methodology for intelligent systems design and testing. In E. Messina and A. Meystel (Eds.), *Proc. of PERMIS, Gaithersburg*. Proceedings of the 2002 PerMIS Workshop.

IEEE (1998). IEEE standards for software verification and validation. IEEE Standard 1012. Washington, DC.

Jennings, N. and M. Wooldridge (1998). Applications of Intelligent Agents. In N. Jennings and M. Wooldridge (Eds.), *Agent Technology : Foundations, Applications, and Markets*. Springer.

Jennings, N. R., K. Sycara, and M. Wooldridge (1998). A Roadmap of Agent Resarch and Development. *Autonomous Agents and Multi-Agent Systems 1*(1), 275–306.

Kinny, D., M. Georgeff, and A. Rao (1996). A Methodology and Modelling Technique for Systems of BDI Agents. In W. Van de Velde and J. Perram (Eds.), *Agents Breaking Away*, Volume 1038 of *LNAI*, pp. 56–71. Springer.

Kitano, H., S. Tadokor, H. Noda, I. Matsubara, T. Takhasi, A. Shinjou, and S. Shimada (1999). Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *Proc. of the IEEE Conference on Systems, Men, and Cybernetics*.

Kitano, H., M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada (1997). The RoboCup Synthetic Agent Challenge 1997. In *International Joint Conference on Artificial Intelligence IJCAI'97*.

McCarthy, C. and M. Pollack (2002). A plan-based personalized cognitive orthotic. In *AIPS-2002*.

McLean, T. and R. Fujimoto (2000). Repeatability in real-time distributed simulation executions. In *14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, pp. 23–32.

Montemerlo, M., J. Pineau, N. Roy, S. Thrun, and V. Verma (2002). Experiences with a mobile robotic guide for the elderly. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada. AAAI.

Montgomery, T. and E. Durfee (1990). Using MICE to Study Intelligent Dynamic Coordination. In *Second International Conference on Tools for Artificial Intelligence*, Washington, DC, pp. 438–444. Institute of Electrical and Electronics Engineers.

Peraire, C., S. Barbey, and D. Buchs (1998). Test selection for object-oriented software based on formal specifications. In *PROCOMET*, pp. 385–403.

Pollack, M. (1996). Planning in Dynamic Environments: The DIPART System. In A. Tate (Ed.), *Advanced Planning Technology*. AAAI.

Pollack, M., L. Brown, D. Colbry, C. McCarthy, C. Orosz, B. Peintner, S. Ramakrishnan, and I. Tsamardinos (2003). Autominder: An intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*.

Pollack, M., C. McCarthy, S. Ramakrishnan, I. Tsamardinos, L. Brown, S. Carrion, D. Colbry, C. Orosz, and B. Peintner (2002). Autominder: A planning, monitoring, and reminding assistive agent. In *Seventh International Conference on Intelligent Autonomous Systems*.

Pollack, M. E. and M. Ringuette (1990). Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *AAAI-90*, Boston, MA, pp. 183–189.

Riley, P. and G. Riley (to appear). SPADES, a distributed agent simulation environment with software-in-the-loop execution. In *Winter Simulation Conference 2003*.

Sargent, R. G. (1999). Validation and verification of simulation models. In *Winter Simulation Conference*, pp. 104–114.

Schattenberg, B. and A. Uhrmacher (2001). Planning Agents in James. *Proceedings of the IEEE 89*(2), 158–173.

Schütz, W. (1993). *The testability of distributed real-time*

systems. Kluwer Academic Publishers, Boston / Dordrecht / London.

Uhrmacher, A., P. Fishwick, and B. Zeigler (Eds.) (2001). *Special Issue: Agents and Simulation: Exploiting the Metaphor*, Volume 89 of *Proceedings of the IEEE*.

Uhrmacher, A. and K. Gugler (2000). Distributed, Parallel Simulation of Multiple, Deliberative Agents. In *Parallel and Distributed Simulation Conference PADS'2000*, Bologna. IEEE Computer Society Press.

Uhrmacher, A. and M. Krahmer (2001). A Conservative, Distributed Approach to Simulating Multi-Agent Systems. In E. Kerckhoffs and M. Snorek (Eds.), *Proc. European Multi-Simulation Conference*, San Diego, pp. 257–264. SCS.

Uhrmacher, A., M. Röhl, and B. Kullick (2002). The role of reflection in simulating and testing agents: An exploration based on the simulation system James. *Applied Artificial Intelligence 16*(9-10), 795–811.

Weyuker, E. J. (1998, September/October). Testing component-based software: A cautionary tale. *IEEE Software 15*(5), 54–59.

Wooldridge, M. and P. Ciancarini (2000). Agent-oriented software engineering: The state of the art. In P. Ciancarini and M. J. Wooldridge (Eds.), *First Int. Workshop on Agent-Oriented Software Engineering*, Volume 1957 of *Lecture Notes in Computer Science*, pp. 1–28. Springer-Verlag, Berlin.

Zeigler, B., H. Praehofer, and T. Kim (2000). *Theory of Modeling and Simulation*. London: Academic Press.

AUTHOR BIOGRAPHIES

JAN HIMMELSPACH holds an MSc in Computer Science from the University of Koblenz. His research interests are on developing methods for agent-oriented modeling and simulation, with a focus on effective and efficient simulation mechanisms and interaction patterns between simulation and software agents. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock.

MATHIAS RÖHL holds an MSc in Computer Science from the University of Rostock. His research interests are on developing methods for agent-oriented modeling and simulation and their application to sociological, biological and software systems. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock.

ADELINDE M. UHRMACHER is an Associate Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. Her research interests are in modeling and simulation methodologies, particularly agent-oriented modeling and simulation and their applications. Web pages of authors can be found at: www.informatik.uni-rostock.de/mosi