Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SIMULATION FOR TESTING SOFTWARE AGENTS – AN EXPLORATION BASED ON JAMES

Jan Himmelspach
Mathias Röhl
Adelinde M. Uhrmacher

Department of Computer Science
University of Rostock
Albert-Einstein-Str. 21
Rostock, M/V, 18059, GERMANY

ABSTRACT

Agents are software systems aimed at working in dynamic environments. Simulation systems can be used to provide virtual environments for testing agents. The software to be tested, the objective of the simulation study, and the stage of the agent software development influences both: the environmental models used for testing and the mechanisms that synchronize the execution of agents and simulation. A clear distinction between model and simulation layer, and a modular design of the simulation system support the required flexibility. Based on the simulation system `James` (a Java based Agent Modeling Environment for Simulation) and two agent applications we will explore, how interfaces between virtual environments and software agents can be explicitly specified at the modeling level and suitable mechanisms for synchronization might be chosen on demand.

1 INTRODUCTION

Agents are software systems that are aimed at working autonomously in dynamic and uncertain environments (Jennings et al. 1998). To construct agents means to develop software that is able to successfully accomplish specified tasks in an environment which changes over time. To support the development of software agents is the goal of agent-oriented software engineering (Ciancarini and Wooldridge 2001).

The variety of ingredients of multi-agent systems, e.g. concurrent objects, Artificial Intelligence methods, and social structures are recognizable in approaches for specifying and developing agents and are responsible for their diversity. Extensions of UML can be found (Odell et al. 2003), same as logic approaches which are particularly aimed at designing rational agents and pruned for reasoning about changing beliefs (van der Hoek and Wooldrige 2003). Other approaches suggest to develop communities of agents by defining social norms and regulations. Thereby the identity of agents is fully specified by their social role in the community (Ryan and Schobbens 2002). All of those approaches use models, however rather different ones for rather different purposes and for different stages of the development process. Whereas object-oriented approaches support the automatic transformation into implementation, logic-based specifications are aimed at supporting the verification of certain properties and thus a static analysis of the software.

In contrast to static analysis, the dynamic analysis of software requires the execution of software. Testing activities support quality assurance by gathering information about the software being studied. Studies indicate that testing consumes easily 50 % of the costs of software development (Harrold 2000). This percentage might even be higher if the software is safety critical or, as in developing agents, the software development process has an intrinsically experimental and explorative nature. "the development of any agent system — however trivial — is essentially a process of experimentation" (Wooldridge and Jennings 1998). Surprisingly, only little work has been done so far on developing methods for testing agents (Dam and Winikoff 2003).

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. Same as the functionality of real-time systems and embedded systems, the functionality of agent systems can not be evaluated based on one time point only. 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. Environment models are used to generate the different test cases dynamically during simulation, including specific interaction patterns and time constraints (Schütz 1993, p.23).

Often test cases are based on and sometimes even automatically generated from software requirements, source-code statements, and module interfaces (Peraire et al. 1998). Due to their typically complex dynamic environment such an automatic generation of dynamic models as test cases for agent systems is difficult to imagine. In addition to the environment the agent is supposed to work in, the stage of the agent development process and the objective of the simulation study will necessarily affect modeling. Typically simulation is employed for behavioral testing, thus rather late in the software development process. However, software testing should start as early as possible in the development cycle — "the earlier a bug is discovered the cheaper the correction" (Beizer 1995, p. 11). The different stages in developing agents require different models that embed the agent. The environment model(s) must be easily adaptable to provide the required granularity and to complement the software agent as far as it has been developed.

As testing in general, simulation cannot show the absence of faults — it can only show its presence (Myers 1979) and the latter only, if the models are valid. The validity of the environmental models will be crucial, independently whether abstract models of agents are experimentally evaluated (Wolpert and Lawson 2002), single agent modules are embedded for testing (Schattenberg and Uhrmacher 2001), or entire agent systems are plugged into the virtual environment (Pollack 1996). Validity is a relation between model, system, and the objective of the simulation study. In accordance to regular testing, a set of simulation studies has to be executed to test the software agent under normal circumstances, to explore boundary cases, and to confront software agents with unexpected situations.

Thus, during the development of software agents, a variety of environment models will be needed including the model that realizes the interface towards the agent software. The type of the agent software, the objective of the simulation study, and the stage of the development process, e.g. whether single modules or entire agents with their own thread of control and possibly with their own clock are tested, influence the suitability of execution mechanisms. The different and, during agent development, also changing requirements have to be met by a flexible model and simulator design.

2 APPLICATION SCENARIOS: MOLE AND AUTOMINDER

The following two application scenarios shall illustrate the necessity to offer different types of interfaces and simulation mechanisms to the agent developer. Our exploration will be based on the simulation system `James`, a Java based Agent Modeling Environment for Simulation (Uhrmacher 2001). A formalism which extends `Devs` by means for reflection, time models, and peripheral ports underlies `James`. Like all simulation systems that are based on `Devs` or its extensions, `James` allows a hierarchical, modular model design and clearly distinguishes between model and simulator. Via reflection models can change their own behavior, composition, and interaction pattern during simulation. *Time models* and *peripheral ports* enable models to interact with externally running software. Time models can be used to translate external resource consumption into simulation time. Via the peripheral ports a model exchanges information with externally running software. Messages to be sent to the software are put into the peripheral output ports and messages directed from the software to the model pass through the peripheral input ports. Peripheral ports form an extension of a model's state from the point of view of transition, output, and time advance functions. The peripheral input ports are read by all of those and the peripheral output ports are charged by the transition functions. This is in contrast to the work of (Cho et al. 2001) where events sent by external software are treated similar to external `Devs`-events, i.e. read by the external transition and sent by the lambda function. The conversion towards `Devs`-messages happens outside of the model. In `James` the interface between simulation and external software is encapsulated within a model.

2.1 Testing Mole Agents with Representatives

`Mole` is a Java-based mobile agent system (Baumann et al. 1997). Locations offer certain services to the agent and represent the source and destination of moving agents. `Mole` agents are equipped with a set of methods, e.g. for migrating, remote procedure calls (RPC), sending and receiving messages, and for handling the individual life cycle. In addition, `Mole` agents can use the entire functionality of `Java`, only constrained by the security model employed.

The life of a `Mole` agent starts in the moment a location initiates the creation of an agent, which includes activating the start method. Whereas the start method runs exactly once, several messages and calls can arrive at the same time. This requires handling several concurrent computation processes. In `James` a `Mole` agent is represented as one core model surrounded by models that represent its running or waiting threads. These simulation models that are called the representative of a `Mole` agent form the interface towards the externally running `Mole` agent.

An example appears in Figure 1. The core agent models are `Agent1` and `Agent2`. The satellite `Start` forms the representative of the running start thread of *Agent1* and the satellite `RPCgetPrice` forms a representative of the thread that has been created by calling the procedure *getPrice* of *Agent2*.

During simulation, the representative reflects the agent's state and behavior. Whereas the agent core model repre-

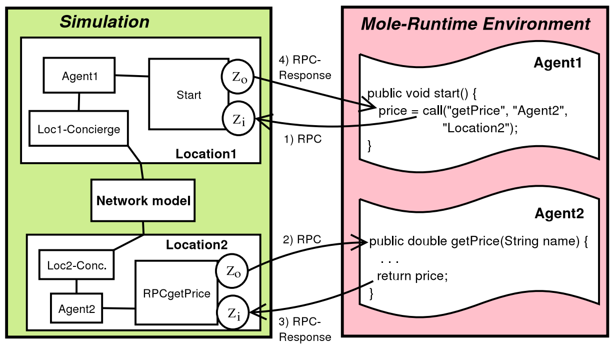

Figure 1: How Mole and `James` are Intertwined

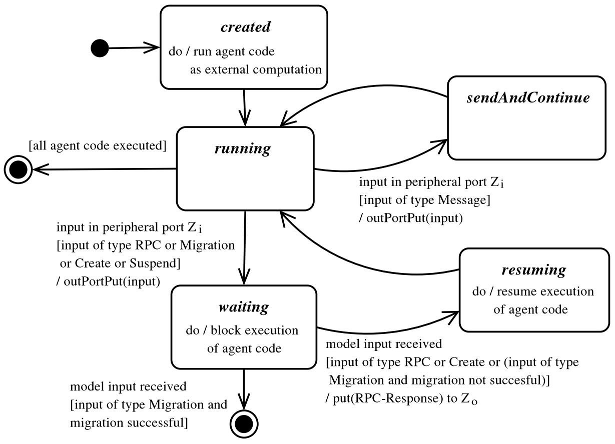
created
do / run agent code as external computation
running
sendAndContinue
[all agent code executed]
input in peripheral port Zi [input of type Message] / outPortPut(input)
input in peripheral port Zi [input of type RPC or Migration or Create or Suspend] / outPortPut(input)
resuming
do / resume execution of agent code
waiting
do / block execution of agent code
model input received [input of type RPC or Create or (input of type Migration and migration not succesful)] / put(RPC-Response) to Zo
model input received [input of type Migration and migration successful]

Figure 2: The Representative of a Thread of a `Mole` Agent as a Statechart

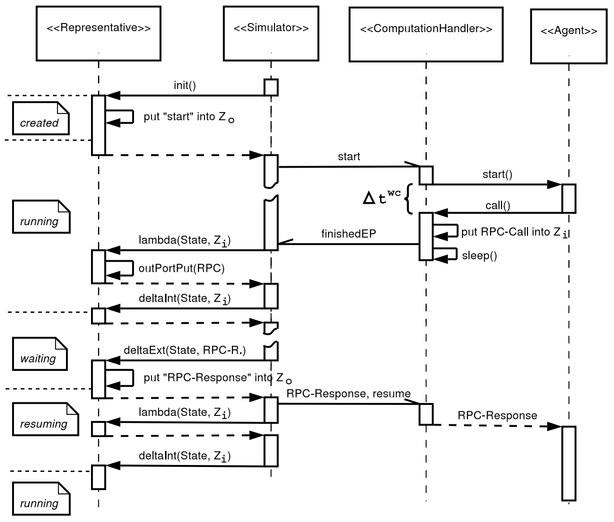

Figure 3: Interaction Between Simulation, Representatives and a `Mole` Agent

sents the central focus of control, its "satellites", e.g. the model `Start`, provide the interfaces to the agent's running processes. Each of the satellites forms a representative of a running or waiting thread of a `Mole` agent. Together with the central agent model they form the representative of an entire `Mole` agent. If the start method of the `Mole` agent encounters a remote procedure call, this call is put into the peripheral input port of the satellite and the thread is suspended. With this the call has entered the virtual environment. In response to the input in its peripheral input port the satellite will forward the remote procedure call to the core agent model and change its state from running to waiting. At the time it receives the result of the remote procedure call via its input port, it will change to resuming. After the external thread has been resumed, the satellite will change to the state running again. As shown in the Statechart in Figure 2, the transitions between the different states are triggered by the flow of time, e.g. `Resuming` to `Running`, by inputs arriving from other models, e.g. from `Waiting` to `Resuming`, or by inputs arriving from the externally running software via the peripheral input port Z_i, e.g. from `Running` to `Waiting`.

Figure 3 shows the interaction between a `James` model, i.e. a `Start` satellite, its `Simulator`, the `ComputationHandler`, and the `Agent`. At the moment the agent core model receives the "start up" notification, it will create the satellite `Start`. With the creation of the `Start` satellite, its simulator is created which will execute the initialize method of the model and start the external computation code. With this the "start up" notification has entered the `Mole` runtime environment.

Messages between simulation and agents are exchanged in time-stamp order. Agents do not have an own simulation clock. Thus, the experiment in the virtual environment, which takes place in virtual (simulation) time, is controlled solely by the simulation which uses a time model to translate the resource consumption, e.g. in wall clock time, of the externally running agent into simulation time. To keep track of the externally running agents and to let simulation and externally running software agents proceed concurrently is the task of the `ComputationHandler`.

Methods in `Mole` are not simply executed as `Java` methods but reflected to make sure that the execution adheres to the security policy. Methods of the `Mole` API which constitute the interface between `Mole` agents and their runtime environment have only to be slightly changed to redirect calls and messages to the simulation system. A concrete agent implementation does not require any changes to be run in the virtual environment. Besides testing performance issues, e.g. comparing migration and remote procedure call, the functionality of single agents in different network environments and agent societies, e.g. confronting the agent with cooperative or defective behavior, can be analyzed.

2.2 Autominder

Autominder is a software system to monitor the activities of elderly and to remind elderly if they forget or confuse certain activities (Pollack et al. 2003). The system shall work on a mobile, autonomous robot — a nursebot. Autominder has an own thread of control and communicates with the robot through a socket. Autominder sends text strings to the robot to be delivered to the client and the robot sends messages containing interpreted sensor information to Autominder.

Both planning and time plays a crucial role in this scenario. Autominder keeps track of the activities of the elderly and tries to remind the elderly in a timely, not annoying, and effective manner. Many activities are scheduled for certain times of the day. The robot has to remind the elderly in time if these activities are crucial for the elderly's health. Therefore, Autominder frequently accesses wall clock time. If Autominder is truly tested the interaction will happen in wall clock time, such that each test run would require a day. A significant evaluation of the effectiveness of Autominder would likely take at least a month.

In the Mole experiment the agent and its currently running thread are described as explicit models, which serve as interfaces. In addition they allow to inspect the internal behavior of the agents during simulation. These representatives support a kind of gray box testing. In contrast, the intention of our experiments with Autominder is behavioral testing or black box testing.

Coupling Autominder and the virtual environment, which currently comprises four models, i.e. the Robot, the Elderly, the Caregiver and the Environment (Figure 4), is done by utilizing the robot model as an interface to loosely couple the agent software Autominder and the simulation. The virtual environment employs no representative of an Autominder model and thus does not support the inspection of internal behavior of Autominder. The objective of ongoing experiments is to test the behavior and the adaptation strategies of Autominder with different types of elderly. 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 is 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.

In Figure 5 the robot asks the environment for sensory information to forward it to the Autominder software. After some time the robot will receive the response of Autominder and will forward it to the elderly. During interaction of James with Mole the ComputationHandler was responsible for managing the synchronization between simulation and agents according to the employed time model. As Mole agents and their methods are invoked by the simulator the ComputationHandler keeps track of all externally running sources and, during their execution, forces the otherwise unpaced simulation to advance in synchrony with the resource consumption. In the scenario with Autominder the interaction between simulation and Autominder occurs directly and events are processed in receive order, which obviates the need for a controlling and monitoring unit like the ComputationHandler.

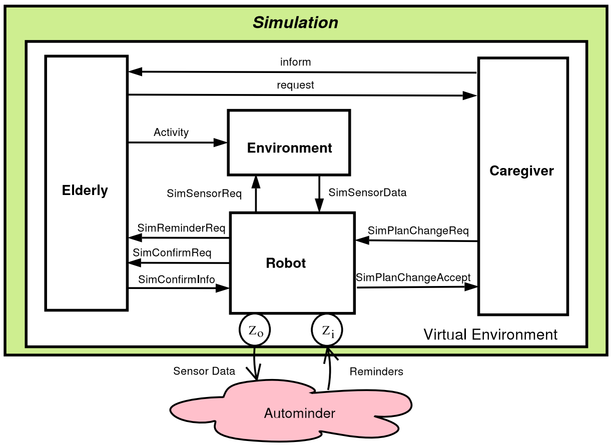

Figure 4: Coupling Autominder to a Virtual Test Environment

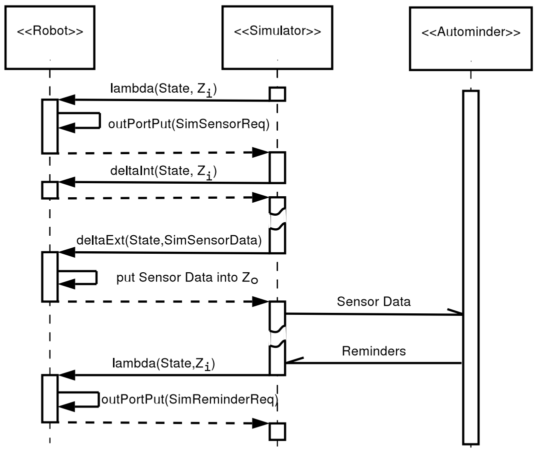

Figure 5: The Interaction Between Simulation and Autominder

3 INTERACTION BETWEEN SIMULATOR AND AGENTS

The interface between agents and simulator is described by models. They serve as interfaces towards externally running software. Peripheral ports and time models are utilized to specify the interaction. Executing the model according to this specification and the given initial situation is the task of a simulator. During the design of agents a variety of models will be employed. Similarly, the different stages of designing agents have different requirements when it comes to executing the simulation experiment. Therefore, `James` contains a variety of simulators which can be distinguished depending on whether the simulation itself shall be executed in a distributed or non distributed environment, the simulation shall proceed paced or non-paced, and whether the simulation shall process events in receive order or in time stamp order. Even though the choice of a certain simulator should not have an effect on the results of the experiment, it has a definite effect on the efficiency of the simulation as well as on the set-up of the testing, e.g. whether and how the software to be tested has to be instrumented for this purpose.

Simulation and agents are executed in wall clock time. However, wall clock time forms not necessarily the basis of interaction between both. In simulation we distinguish between physical time, simulation time and wall clock time. Whereas simulation time and physical time are connected by a semantic relation, e.g. one tick in simulation time refers to one minute in physical time, wall clock time is not necessarily related to either of both. During the execution of the simulation the wall clock time advances, more or less independently of and even alternating, with simulation time. Only in paced simulation the simulation time advances in synchrony with wall clock time (Fujimoto 2000).

Figure 6 shows how simulation time and wall clock time advances given a discrete event simulator. The version a) shows an unpaced version in which a discrete event simulation runs as fast as possible, b) and c) are both paced. In paced simulation, independently whether it is scaled (c) or not scaled (b), the relation between simulation and wall clock time has to be carefully observed to avoid a situation where the simulator lags wall clock time (see c).

3.1 Unpaced Simulation

One unpaced `James` simulator realizes a distributed simulation system which executes simulation events concurrently that occur at the same simulation time. Thus, within the simulation the concurrency is rather limited. However, the simulator has been developed to support an efficient testing of a few deliberative agents by splitting simulation and externally running agents into different threads and letting them proceed concurrently (Uhrmacher and Gugler 2000).

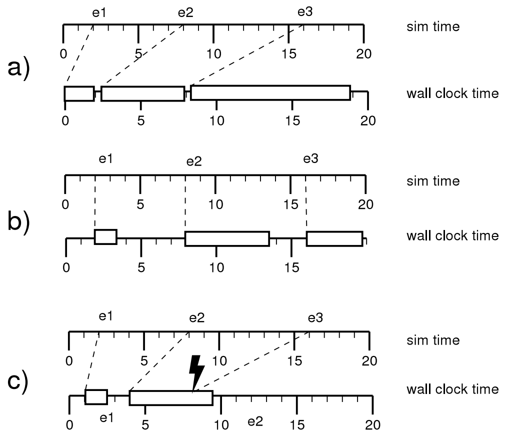

Figure 6: Advancing Simulation Time and Wall Clock Time

Each time the simulator wishes to advance to the time of next event, first all externally running processes are asked whether it is safe to proceed (Figure 7). To process events in time-stamped order, the event from the agent is labeled with a simulation time which determines when the results of the agents shall return into the simulation. Therefore, a time model is employed, which transforms the resources that the agent consumed into simulation time. It defines the relation of resource consumption and simulation time locally for each model.

$$t = t_{Start} + TimeModel(ConsumedResources)$$

Often the already consumed resources refer to wall clock time and in this case the processing of events is scheduled to occur depending on the advance of wall clock time.

$$t = t_{Start} + TimeModel(t^{wc}_{now} - t^{wc}_{start})$$

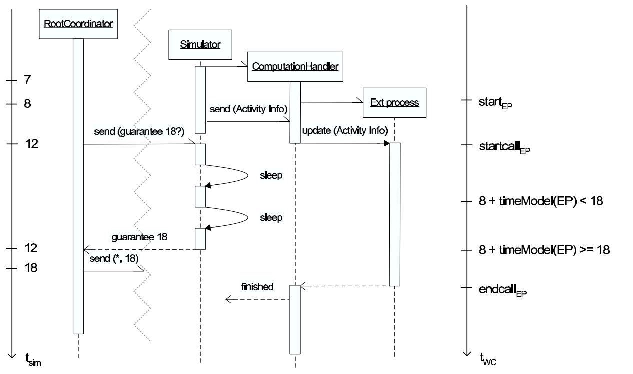

Figure 7: The Interaction Between External Processes and an Unpaced, Conservative Simulator Combined with an Exchange of Messages in Time-Stamp Order

As long as the consumed simulation time of the externally running agent is less than the time of next event in the simulator, the simulator will wait. An optimistic simulator could continue and roll back if agents deliver an event in the simulator's past. However, any message that the simulator sends to the agent software has to be treated as an I/O operation and thus may not be rolled back. Not the entire simulation proceeds in synchrony to wall clock time. The time models are defined locally and during episodes in which no external agent is running, the simulation will execute as fast as possible.

If the simulator processed events from agents in receive order, the simulator would only check whether there are any inputs from the agents before advancing simulation time. If messages had arrived, it would process the received events at the time of the last calculated event as the event from the agents occurred somewhere in between the last and the next scheduled event. The combination of an unpaced simulator and the processing of agent events in receive order is used for competitive rather than analytical purposes. An as fast as possible discrete event simulation jumps from one event to the other thereby neglecting the simulation time that lies in-between. The same consumption of wall clock time by the agent would result in quite different reaction times measured in the virtual environment during a simulation run. If agents are competing with each other in a virtual environment, as it is the case in `RoboCup` (Kitano et al. 1999), it is only important that the time pressure for each agent is the same; whether the time pressure varies throughout one experiment is of less importance.

Another criterion which influences the set-up of the experiment is whether the simulator is controlling the execution or whether simulator and agent run more or less independently. As shown in Figure 7, the unpaced simulator invokes the agent, respectively modules of the agent, and keeps track of all external processes: when external processes have been started, which processes have been finished, and at which simulation time the completion shall be scheduled (Uhrmacher and Krahmer 2001). The external processes can represent entire agent systems, or single threads of agents like the `start` and `getPrice` thread in the `Mole` scenario. To let agent and simulation run independently and exchange events in time-stamp order requires to equip the agent software with an own simulation clock and to provide means to advance the simulation time.

3.2 Paced Simulation

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. Therefore, each simulator has to block until its local virtual time has reached the required wall clock time. To relate simulation and wall clock time a simple time model is used. This time model globally relates wall clock time and simulation time for all logical processes.

$$t = t_{start} + Scale * (t^{wc}_{now} - t^{wc}_{start})$$

A scaled paced simulator allows to let the simulation, e.g., run twice or half as fast as wall clock time. 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. Therefore $t_{start}, t^{wc}_{start}$ refer not necessarily to the starting point of the simulation but to the starting point of pacing simulation with a specific scaling factor. To speed up simulation should be done within the boundary of resources needed for processing events, even though a lack of slack time during the fast cycles will possibly be compensated during the episodes of slower processing.

The paced simulator (Figure 8) has been developed by adapting the distributed real-time simulator introduced by Zeigler and Cho for `Devs` models. First the simulator waits for the wall clock to elapse or an external or peripheral event to arrive. In addition, the simulator waits for a small period of time, which is defined by the user to let further events arrive (Cho 2001). Every time an event takes place, the peripheral ports are charged with the event sent by the agent. Generally, the arrival of an event from an external process is associated with an internal or confluent transition in the model. With both transitions the production of an output is associated. The state is updated, which also might include charging the peripheral output port and thus sending messages to the agent software. Simulators and external software exchange messages in an asynchronous manner.

```
while simulation not yet finished
  $t = t_{start} + Scale * (t^{wc}_{now} - t^{wc}_{start})$
  blockUntil $(t \geq t_{next}) \vee$
    $externalEventFromModel \vee peripheralEventFromAgent$
  WaitToCheckForFurtherEvents()
  charge $z_i$
  if $t = t_{next} \vee peripheralEventFromAgent$ then
    send $(\lambda(s, z_i))$ to parent
    if $externalEventFromModel$ then
      $(s, z_o) = \delta_{con}(s, xb, z_i)$
    else
      $(s, z_o) = \delta_{int}(s, z_i)$
    endif
  else
    $(s, z_o) = \delta_{ext}(s, t - t_{last}, xb, z_i)$
  endif
  ...
  $t_{last} = t$
  $t_{next} = t_{last} + ta(s, z_i)$
  flush $z_i$
end
```

Figure 8: Extract of a Paced Distributed Simulator in `James`

In this case all simulators run in real-time and process events in receive order. A general problem with this kind of real-time simulation is repeatability. (Bacon and Goldstein 1991) classified non-determinism arising from input-data, system calls, and interrupts. Non-determinism arising from input data can be distinguished whether the input has been sent by logical processes, by externally running software, or by other sources, like human operators (McLean and Fujimoto 2000).

Whereas in the context of `Mole` and for testing different planning agents (Uhrmacher and Gugler 2000) parallel, distributed simulators have been employed, for testing `Autominder` in the current virtual environment a sequential simulation suffices. Thus, one source of non-determinism, which is induced by messages that are passed between and processed in receive order by logical processes of distributed, parallel simulators (Figure 9), is avoided. However, the other sources of non-determinism still remain.

```
while simulation not yet finished
  $t = t_{start} + Scale * (t^{wc}_{now} - t^{wc}_{start})$
  blockUntil $(t \geq t_{nextEvent}) \vee$
    $peripheralEventFromAgent$
  if $peripheralEventFromAgent$ then
    for all simulators
      charge peripheral input port $z_i$
    $peripheralEventFromAgent = false$
  calculate Imminents
  for all simulators $\in$ Imminents
    execute $(\lambda(s, z_i))$ and propagate to input ports
      of Influencees and update Influencees
  for all simulators $\in$ Imminents
    if $externalEventFromModel$ then
      $(s, z_o) = \delta_{con}(s, xb, z_i)$
    else
      $(s, z_o) = \delta_{int}(s, z_i)$
    endif
  for all simulators $\in$ Influencees $\setminus$ Imminents
    $(s, z_o) = \delta_{ext}(s, t - t_{last}, xb, z_i)$
  ...
  for all simulators $\in$ Imminents $\cup$ Influencees
    $t_{last} = t$
    $t_{next} = t_{last} + ta(s, z_i)$
  $t_{nextEvent}$ = minimum of next events of simulators
  for all simulators
    flush peripheral input port $z_i$
end
```

Figure 9: Extract of the Sequential Paced Simulator in `James`

The sequential simulation holds all information about the simulators, which are reduced to data structures and are no longer active threads — their associated atomic models, the coupled models they belong to, and the coupling between models. The simulation waits until the time of next event is reached or messages of an external source have arrived. These messages are the only ones still processed in receive order. The peripheral input ports are charged with the messages from the agents, and the corresponding output functions and transition functions are invoked. The time of next event is calculated and the peripheral ports are flushed. Whereas the sequential simulator suffices in our current `Autominder` scenario, a distributed paced simulation system will be necessary for an efficient testing of most agent applications, particularly mobile agent applications. Testing mobile agents and their strategies, e.g. (Küpper and Park 1999), requires often valid network models and thus, fast and efficient parallel, distributed simulation strategies. Currently we are designing a paced distributed simulation system which uses time-stamped messages. The underlying concept is similar to (McLean and Fujimoto 2000) in utilizing time stamps for reducing non-determinism.

Scaling a paced simulation that exchanges events in receive order time with agent software in general means to lessen or to amplify the time pressure for the agent software. In the case of `Autominder`, whose activities depend on wall clock time, the call to the internal clock is re-directed to a virtual clock, which advances time according to the scaling factor used in the simulation to speed up or to slow down the experiments. This can also be done during the simulation run.

If a scaled paced simulation exchanged events in time-stamp order the local time models used to transform wall clock time into simulation time would have to be carefully selected. A simultaneous usage of different time models, e.g. locally within each model as utilized in the `Mole` scenario, or within externally running agents — the latter would be easy to realize in the `Autominder` scenario — and globally within the scaled paced simulation, aggravates the interpretation of simulation results.

4 CONCLUSION

The testing of agents by simulation requires different models and also different strategies for synchronization. Whether representatives of the software to be tested are used depends on how early in the designing process the simulation is employed. Representatives can be derived from specifications and lend themselves to testing specific aspects of the agents. Because they can be used to complement the software developed so far and provide insights into the internal functioning of agents and their modules during execution, representatives will be of use to interface simulation and agents early in the design process. However, when simulation is applied to test software, most of the time it will support so called behavioral testing. This black box testing, often executed by a third party, usually represents 35 to 65 percent of all testings whereby these percentages are even higher for object oriented programs (Beizer 1995, p. 11) and may even be topped by agent software. If the entire system is tested it will be rather an environment model than a representative that will serve as an interface between simulation and agent system. Thus, during the development process the coupling between simulation and agent systems might be gradually loosened.

In the beginning of developing the agent system not only the environment but also part of the agents will be modeled. Therefore, an as fast as possible execution and the use of local time models might support the early stages of designing agents best. Later the agent software gains weight and autonomy in the experimental setting. Real-time executions of simulation and treating simulation and agents as equal partners might provide the answer to this changed perspective. However, it is not the case that paced simulation is exclusively used in later phases and unpaced simulation in earlier phases. Thus, the required flexibility refers not only to the modeling layer, but will permeate all layers of simulation systems which are aimed at supporting the design of software agents.

ACKNOWLEDGMENTS

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

REFERENCES

Bacon, D., and S. Goldstein. 1991. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Volume 26 of *ACM SIGPLAN Notices*, 194–206.

Baumann, J., F. Hohl, K. Rothermel, and M. Strasser. 1997. Mole-Concepts of a mobile agent system. *WWW Journal - Special Issue on Applications and Techniques of Web Agents* 1 (3): 133–137.

Beizer, B. 1995. *Black-Box Testing*. John Wiley & Sons, Inc.

Cho, Y. 2001. *RTDEVS/CORBA: A Distributed Object Computing Environment for Simulation-Based Design of Real-Time Discreet Event Systems*. Ph. D. thesis, Electrical and Computer Engineering Dept., University of Arizona.

Cho, Y., B. Zeigler, and H. Sarjoughian. 2001. Design and implementation of distributed real-time DEVS/CORBA. In *IEEE Systems, Man, and Cybernetics Conference*. Tucson.

Ciancarini, P., and M. J. Wooldridge. (Eds.) 2001. *Agent-Oriented Software Engineering*, Volume 1957 of *Lecture Notes in Computer Science*. Springer.

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.

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

Harrold, M. J. 2000. Testing: A roadmap. In *ICSE - Future of SE Track*, 61–72.

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.

Kitano, H., S. Tadokoro, 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*.

Küpper, A., and A. Park. 1999. Realizing Nomadic communication with mobile agents: Strategies and their evaluation. In *Telecommunications Information Networking Architecture Conference*.

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

Myers, G. 1979. *The Art of Software Testing*. John Wiley & Sons, Inc.

Odell, J., H. Parunak, M. Fleischer, and S. Brueckner. 2003. Agent UML: A formalism for specifying multiagent software systems. In *Agent-Oriented Software Engineering III*, ed. F. Giunchiglia, J. Odell, and G. Weiss, Volume 2585 of *Lecture Notes in Computer Science*, 16–31. Springer.

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

Pollack, M. 1996. Planning in dynamic environments: The DIPART system. In *Advanced Planning Technology*, ed. A. Tate. 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*. to appear.

Ryan, M., and P.-Y. Schobbens. 2002. Agents and roles: Refinement in alternating-time temporal logic. In *Intelligent Agents VIII: Agent Theories, Architectures, and Languages*, ed. J. Meyer and M. Tambe, Volume 2333 of *Lecture Notes in Artificial Intelligence*, 100–114. Springer-Verlag.

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. 2001. A system theoretic approach to constructing test beds for multi-agent systems. In *A Tapestry of Systems and AI-based Modeling & Simulation Theories and Methodologies: A Tribute to the 60th Birth-*

day of Bernard P. Zeigler, ed. F. Cellier and H. Sarjoughian, Lecture Notes on Computer Science. New York: Springer.

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 *Proc. European Multi-Simulation Conference*, ed. E. Kerckhoffs and M. Snorek, 257–264. San Diego: SCS.

van der Hoek, W., and M. Wooldrige. 2003. Towards a logic of rational agency. *Journal of Autonomous Agents and Multi-Agent Systems* 11 (2): 133–157.

Wolpert, D., and J. Lawson. 2002. Designing agent collectives for systems with markovian dynamics. In *AAMAS 2002: Autonomous Agents and Multi-Agent Systems*.

Wooldridge, M., and N. Jennings. 1998. Pitfalls of agent-oriented development. In *Proceedings of the 2nd International Conference on Autonomous Agents*, 385–391.

AUTHOR BIOGRAPHIES

JAN HIMMELSPACH holds a 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 possible 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 a 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>`