Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

"PLUG AND TEST" – SOFTWARE AGENTS IN VIRTUAL ENVIRONMENTS

Adelinde M. Uhrmacher

Department of Computer Science
University Rostock
D-18051 Rostock, GERMANY

Bernd G. Kullick

Faculty of Computer Science
University Ulm
D-89069 Ulm, GERMANY

ABSTRACT

`James` - A Java Based agent modeling environment for simulation has been developed to support the compositional construction of test beds for multi-agent systems and their execution in distributed environments. The modeling formalism of `James` imposes only few constraints on the modeling of agents and facilitates a "plug and test" with pieces of agent code which has been demonstrated in earlier work. However, even entire agents can be run in `James` as they are run in their run-time environment. The integration of agents as a whole is based on model templates which serve as the agents' interface and representative during the simulation run. The effort which is put into defining model templates for selected agent systems obviates the need for the single agent programmer to get acquainted with the underlying modeling and simulation formalism. Instead the agent programmer can compose the experimental frame and test the programmed agents as they are. The approach is illustrated with agents of the mobile agent system `Mole`.

1 INTRODUCTION

Testing is an obligatory step of each software engineering process and becomes even more important if the development of a software system must be considered as experimental itself. "At the time of writing, the development of any agent system - however trivial - is essentially a process of experimentation. There are no tried and trusted techniques available to assist the developer" (Wooldridge and Jennings 1998). Agent systems typically comprise multiple agents which are bounded for open and dynamic environments. They should be sufficiently flexible to adapt themselves to changed environments and changed functional requirements, and they should be able to do this in time.

The implementation and application of dynamic test scenarios for multi-agent systems require considerable efforts. The virtual environment has to be modeled, as have the interaction between agent and virtual environment. Typically, simulation systems allow to plug in code fragments, or single modules whereas the agent itself is described as part of the model. Other simulation systems treat agents as external source and drain of events. These simulation systems save the user the extra effort to describe the agent in the modeling language of the simulation system. However, they require typically more effort in analyzing the interaction and actions of agents in the virtual world, since neither agents nor their interaction belongs to the modeled world.

Based on a discrete event modeling and simulation formalism for testing multi-agent systems and its concrete implementation `James` (Uhrmacher 2000) we will explore how agents can be plugged and executed in virtual worlds.

2 EMBEDDING AGENTS IN VIRTUAL WORLDS

Testing of agents in simulation systems requires to define and implement the interface between agents and the virtual, dynamic environment including typically, as stated by Hanks and his colleagues (Hanks et al. 1993), the time model. It describes the time the agent will probably need, e.g. to react to changes in its environment or to come up with a detailed plan, when being executed in the target environment.

2.1 Defining Time Models

A variety of different time models have been proposed to capture the temporal dimension of an agent's activity. A constant time model is only employed if the functionality of the agent is not time sensitive or the latter shall not be analyzed. Alternatively, the time needed for deliberation or reaction is sometimes calculated based on the state of the agent. Yet, to foretell the performance of a planning system based on the size and structure of a knowledge base is rather difficult. Therefore, the virtual deliberation time is often modeled as a function of the wall clock time used for executing the deliberation, even though executing the experiment on different machines might affect the performance of the agents influencing the overall result. Counting executed instructions is less susceptible to variations within the execution environments (Anderson 1997), but presumes a "timed" version of the language in which the agent is programmed.

Other test beds for multi-agent systems do not employ an explicit time model, at all. The simulation and agents communicate with each other asynchronously in wall clock time (Itsuki 1995, Saphira Manual 1997). For example the simulator of the environment checks frequently whether the agents have decided on an action that the simulation engine has to take into account, otherwise it proceeds with its own calculation. If the agents are not the only source of dynamics the execution time of agents and the virtual time of the simulation system have to be put into relation which is typically done by slowing down the execution of the simulation system. Due to varying loads on a network, additional noise will be introduced and restrains the control of the experiment.

2.2 The Interface between Agents and Simulation

If the simulation system is not built for one agent type only, functions which realize the interaction of the agent with its normal execution environment are re-implemented or replaced by functions directed to the simulation environment. Agents place timed events on the event queue of the simulation system (Pollack 1996, Anderson 1997) or direct events asynchronously to the ports of the simulation system. For the simulation system an agent forms just an external source and drain of events. If an explicit time model is employed, the agent produces time stamped events which are scheduled as any other events by the simulation system. If agents communicate asynchronously, synchronization means agents and simulation are checking frequently their ports for new information.

Messages and activities of the agents have to be translated into events of the simulated virtual world. Events that are produced by the simulation system and directed to the agent have to be translated into sensory inputs, messages, and calls of methods that are comprehensible by the agent's modules.

Both simulation and agents will keep some kind of representation of the other. Within the virtual world those of the agent's properties are represented that are of global interest, e.g. the position of the soccer player in the field. Depending on the type of agent the agent will maintain an individual view of the virtual world, as it would maintain a representation of the real environment. Agents and test environment are clearly separated and no "model" of the agent exists in the test environment.

To perceive an agent as part of the virtual world, more or less rigorous frames are provided to compose agents based on partly re-using and partly re-implementing an agent's modules, e.g. (Montgomery et al. 1992, Atkin et al. 1998). Thus, they defer the problem of defining the interface between agent and simulation to the modules that are re-used. The agent itself is described within the modeling formalism which more or less favors certain agent architectures.

3 MODELING AND TESTING AGENTS IN JAMES

`James`, a Java-Based Agent Modeling Environment for Simulation (Uhrmacher and Schattenberg 1998, Uhrmacher et al. 2000), constitutes a framework which is aimed at supporting experiments with agents under temporal constraints. Its core libraries provide the means for the description of variable structure models and their distributed, parallel execution. It is not the intention of `James` to provide a reference model for an agent architecture. Instead, it aims at building a firm ground for testing agents and their interaction with dynamic environments, while imposing minimal restrictions on the type of agent or the reasoning mechanism to be tested.

The model design of `James`, resembles that of `Devs` (Zeigler et al. 2000) extended by means for reflection which allows agents to adapt their composition, interaction and behavior pattern (Uhrmacher 2000). In the following section, we will illustrate the model design of `James` by describing earlier experiments in which the model design in `James` was used as a frame to describe agents and in which the user had to plug in agent modules.

Figure 1 illustrates how a planning agent has been described as a time-triggered automaton in `James` and how different functionalities, e.g. updating beliefs, deciding on goals, and the development of plans, have been invoked and thus been tested (Uhrmacher and Schattenberg 1998). If the agent receives an external event in its input port it updates its beliefs and if no plan exists it develops based on its current beliefs, goals, and operators a new one. It determines the time it will become active again. This is the time needed for deliberation or its pure reaction time. After that time the agent will be activated again by the simulation system. The output function will charge the output port with the first entry of the list of intended things to do. Output function and internal transition function form a unity in `James` as they do in `Devs`, so directly after the output function the internal transition function is invoked to update the internal beliefs and things still to do. It also executes that part of the activity which is not sent via the output port. This kind of activity includes e.g. the initiation to change its own interaction structure, to create a new model, to delete itself, or to move itself to a different interaction context. By setting the time advance to infinity the agent signalizes that after having processed this internal event it will wait "forever" for a new input to arrive. In the above example, the actions realize the interface from an agent to its environment. The actions are partly directed to an agent's internal world, e.g. its knowledge base and its set of intentions, those will be interpreted by the planning and belief modules of the agent, part of the actions are directed

```
class StillSimplePlanningAgent extends AtomicModel {

  State deltaExt(State state, double elapsedTime) {
    state.beliefs.update(input, elapsedTime);
    if (intentions.plan == "noOp") {
      state.goals.update();
   state.intentions.update(beliefs,goals,operators);
   state.setTimeAdvance(intentions.deliberationTime);
    } else {
      state.setTimeAdvance(REACTION);
    }
    return state;
  }

  void lambda(State state) {
    Action action = state.intentions.getAction();
    if (state.beliefs.entail(action.pre))
      outPortPut("out",action.outputEffect(state));
    else outPortPut("out", noOp);
  }

  State deltaInt(State state) {
    Action action = state.intentions.getAction();
    if (state.beliefs.entail(action.pre)) {
       action.transitionEffect(state);
       state.intentions.popAction();
       state.setTimeAdvance(INFINITY); }
    else { // re-planning  ...
   state.intentions.update(beliefs,goals,operators);
   state.setTimeAdvance(intentions.deliberationTime);
    }
    return state;
  }
}
```

Figure 1: Extract of the "Still Simple Planning Agent" in `James` (Uhrmacher and Schattenberg 1998)

to the virtual environment, i.e. `James` models and their interpretation of the incoming events.

As in `Devs`, atomic models can be grouped into coupled models. They define the current interaction context of a model. The modular, hierarchical modeling concept facilitates the re-use of components and thus the construction of virtual test environments by composition. The modeling formalism provides a general frame which imposes little constraints on the agent architecture and modules to be tested. However, still the user has to get acquainted with the basic notions of the underlying modeling formalism and the additional effort - in `James` to model agents as time triggered automata - might hold little appeal to agent programmers.

One would like to have both: the ability to execute agents as they are, switching arbitrarily between an execution in the real environment and the virtual test environment. On the other side agents should be an integral part of the experimental setting and should as such be perceivable and controllable rather than function as black boxes loosely interacting with the test environment. One idea to oblige both is to associate a discrete model, a kind of abstract representative of the agent, with the real agent running. The role of the representative is to reflect the relevant behavior of the running agent within the virtual world. The questions to be solved are where do the representatives come from and how are the virtual representative and the agent it is representing interconnected and synchronized.

4 PLUGGING AGENTS INTO JAMES

To interrelate and synchronize the representative and the externally running agent a means for communicating between both is required. These mechanisms do already exist in `James`. They have originally been developed to integrate deliberation processes into simulation runs explicitly and efficiently (Uhrmacher and Gugler 2000). Processes can be started by the atomic model to run concurrently with the simulation and report their results back to the model at a time determined by the employed time model. The results are put into a special port, all `James` models are equipped with, the "peripheral" port. It complements the ports with which atomic models communicate with other models. The peripheral port is charged at a certain simulation time and represents the link from the agent to its representative. To illustrate the approach we choose agents of the mobile agent system `Mole` (Baumann et al. 1997a, 1997b).

4.1 `Mole` Agents

The environment of `Mole` agents comprises engines, which represent the `Mole` runtime system. The engine transforms and forwards messages between the locations and the network. Each engine might comprise a set of locations. They offer certain services to the agent and represent the source and destination of moving agents.

The life of a `Mole` agent (Figure 2) starts in the moment a location or another agent initiates the creation of an agent. To become an active member of an agent's society the preparation method signs responsible. The preparation method is invoked if an agent is created or just awakened after a successful migration. Thereafter, the working phase of an agent starts, which includes invoking the start method, activating the heart beat, and handling incoming messages and calls concurrently. Whereas the start and the heart beat runs exactly once (if at all), several messages and calls can arrive at the same time which require several computation processes to handle them. Also within the agent the user can start additional threads of control. If in one of the several running threads the code encounters a `migrateTo`, the agent prepares itself for its migration and suspends all active threads. This phase will, if the migration succeeds, finally lead to all threads being stopped. If the migration fails the processes are resumed.

`Mole` agents are equipped with a set of methods, e.g. for migrating, RPCs, sending and receiving messages, and for

```
import mole.*;
public class Sputnik extends UserAgent
  implements MobileAgent {
protected LocationName orbit;
protected boolean inOrbit;
public Sputnik() {}
public boolean init(Hashtable hash) {
  String s = (String)parameters.get("Orbit");
  if (s != null) orbit = new LocationName(s);
  else
    return false;
  return true;
}
public boolean prepare() {
  // are we in orbit already?
  if (getCurrentLocation().
        locationName().equals(orbit))
    inOrbit = true;
  else
    inOrbit = false;
}
public void start() {
  if (inOrbit) Engine.out("beep!");
  else
    Engine.out("Sputnik: launching into orbit...");
    migrateTo(orbit);
    Engine.error("Sputnik: launch failed,
                          self-destruct activated.");
    die();
}
public void stop() {
  Engine.out("Sputnik: going down...");
}
```

Figure 2: An Example of a `Mole` Agent <`http://mole.informatik.uni-stuttgart.de/docs/cookbook.html`>

handling the individual life cycle. In addition, `Mole` agents can use the entire functionality of `Java`, only constrained by the security model employed. Agents can comprise a dynamic set of concurrent running or waiting threads and are not restricted to one line of activity.

4.2 The Representative

In modeling `Mole` agents in `James`, we decided on a model, where the agent is represented as one atomic model surrounded by atomic models that represent its running or waiting threads. On demand, the thread models are created and deleted. Thus, a stop of a thread at the `Mole` layer implies the deletion of the corresponding atomic model at the `James` layer. Figure 3 shows an atomic model with its satellites which represent a `Mole` agent with a heartbeat, the start method, and the method "DispatchRPC" running. The representative provides an abstract view of the state and the behavior of the agent. The latter of which is described by piecewise constant trajectories, where each episode is separated from the next by the occurrence of an event.

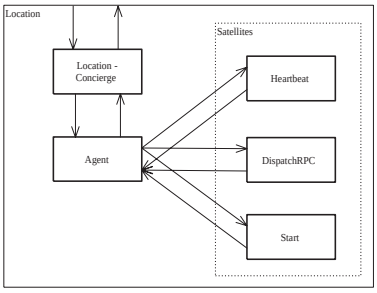

Figure 3: Locations, Agents and their Processes in `James`

The sending and receiving of messages, the departure and arrival of an agent, denote events which are also notable in the interaction with the virtual environment. Other events, e.g. being started or being stopped, highlight "internal" state changes that will influence an agent's reaction to incoming events. These are the events we wish to distinguish in the representative (Figure 4).

Some of the state changes are initiated by incoming events produced by other models, including other agents. Other state changes are initiated by the agent itself by invoking methods which trigger certain transitions and the creation of outputs. The time at which they trigger those transitions depends on the time used to come up with the event and the time model employed. The agent's heart beat is triggered according to the pre-defined frequency.

A subset of the `Mole` API has been re-implemented e.g. migration, RPC, heartbeat, and asynchronous message passing. The re-implemented methods install the relation between a `Mole` agent and its run-time environment, e.g. to send messages with different degrees of reliability `sendUnreliable`, `sendReliable`, `sendMailbox`, calling the method of an agent `call`, a form of suicide `die`, a time triggered activation of the agent `heartbeat`, creating a new agent `createAgent`, and migrating to a new location `migrateTo`. Each of which is defined with a name and a signature as it can be found in the agent class of `Mole` whereas its internal implementation differs by being directed to the virtual environment of `James`.

The methods of the agent are triggered via events that reach the model and which are translated via `Java` reflection into calling concrete methods of the agent or into forwarding concrete message types to the `Mole` agent. For that purpose, the representative holds a reference to the agent as do its satellites. All of the incoming events will either lead to generating an additional real-time process at the `Mole` layer, which implies the creation of a new satellite at the `James` layer, or to resuming (in the case of the RPC) an old process at the `Mole` layer, which is reflected by a state change from waiting to running in the corresponding satellite. The migration of an agent causes all satellites to vanish.

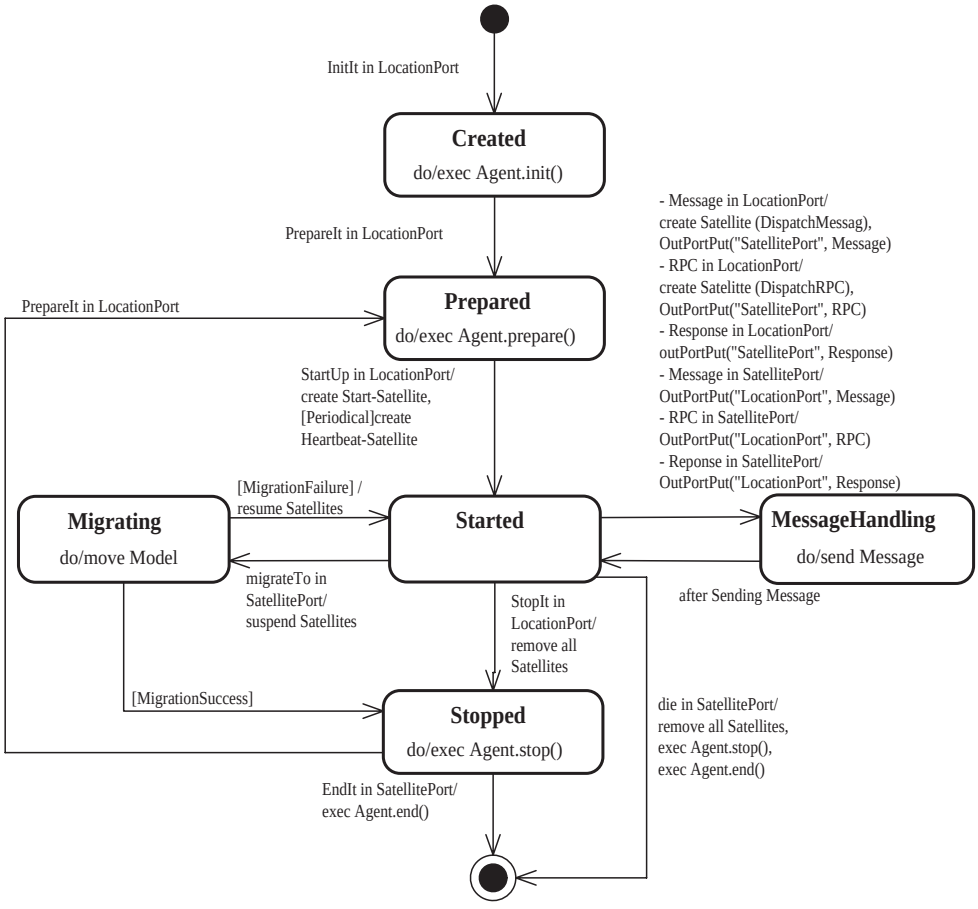

Figure 4: The `James` Core Model of a `Mole` Agent Described as Statechart

The statechart (Figure 4) describes the behavior of the atomic model which represents the agent. After initialization and preparation the agent enters its normal working phase. The start method is invoked. If the agent implements the periodical interface, a heartbeat is started as well. At the moment the location forwards a message to the agent, a dispatch handler is created and started initialized with a reference to the agent and the message received. At the moment a method of the agent is called a satellite is created, initialized with the reference to the agent, the name of the method and its parameters. Thus, all satellites of the representative hold a reference to the same "external" agent as does the representative they surround.

The working phase of an agent is characterized by concurrent threads each represented as a satellite and each in two different states, i.e. `Running`, and `Waiting` (Figure 5). The working phase of those satellites starts with running, that is executing the agent code. If the agent executes one of the asynchronous `Mole` methods, e.g. `sendReliable`, the satellite associated with the thread will notice it in the moment its peripheral port `Z` is charged. The atomic model will fill its own output port with the message, sends the message, and stays in the running phase. If the code

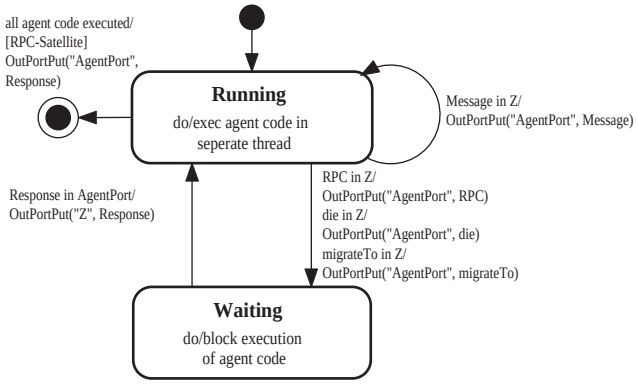

Figure 5: The `James` Model of a Satellite Described as a Statechart

produced a remote procedure call then again the output port of the atomic model is filled, its state changes to waiting.

The peripheral port represents the link from the agent to the simulation whereas the connection from the simulation to the agent is realized by a reference to the agent code from the atomic models' state. The re-implemented methods cause the simulator to fill in the peripheral port and thus, trigger (internal or confluent) state changes within the atomic

model. The methods of the agent are triggered via the "usual" external events which are translated via Java reflection into calling concrete methods of the agent or into forwarding concrete message types to the agent.

The re-implemented API transforms calls, messages, etc. sent by the agent to the virtual environment, into the modeling formalism. Communicating in the other direction necessitates no special re-implementation, since the Java reflection can be used.

4.3 The Environment of Mole Agents

The physical network represents the environment of the Mole system. Compared to many network simulation systems (Cowie et al. 1999), our view of the physical network is currently rather coarse. However, the underlying modeling formalism allows a step-wise refinement of components and supports an inter-operation with other simulation systems (Zeigler et al. 1999): mechanisms which can be exploited for a fine grained network model in the future. Currently, connections and nodes are explicitly represented as atomic models whose efficiency and effectiveness are controlled by a small set of parameters. For example, connections are characterized by bandwidth, reliability, and latency. These attributes influence the throughput of connections: whether and how fast they transport incoming messages to the corresponding output ports.

Mole agents do not directly communicate with the physical network, their messages and calls are conveyed by two other important types of components in the Mole system: engines and locations.

The engine represents the Mole runtime system. It transforms and forwards messages between the locations and the network. Each engine might comprise a set of locations, each of which offers certain services and might embrace several agents. The hierarchy of engines, locations and agents is reflected in a hierarchy of coupled models.

Each location is described by a coupled model and a concierge: an atomic model responsible for welcoming new agents in the location, for forwarding messages to the agent and the engine, and for storing messages directed to agents which are currently abroad. Since coupled models do not have a behavior of their own in James the functionality of a "Mole location" has been encoded in an atomic model, that is the concierge, whereas the aspect of embeddence is described as a coupled model. As is the agent, the location is associated and interacts with a location object of the Mole system.

The engine has a similar structure and, as agents and locations, is connected with a Mole engine. Its concierge provides additional service methods, e.g. creating new names for new agents. The engine represents the gateway to the virtual network, as an engine typically represents the gate for an agent to the physical network. However in this case the connected Mole engine solely interacts with its representative in the simulation.

Thus, all three models, engine, location, and agents do not only represent components of the Mole system and the hierarchical structuring of these components. In addition, they are connected to according Mole objects. Calls to methods and messages are forwarded to the Mole components by using Java reflection. After a time needed for their calculation, their reaction is fed into the virtual environment through the peripheral port. Crucial changes of the Mole engine, location, and agent are reflected within state changes of the introduced James representatives.

5 SIMULATION

One possible execution of multiple plugged-in agents is to forego the need for synchronization in virtual time by installing an asynchronous protocol without explicit time model (Itsuki 1995). Another possibility is to sequentially execute simulations and agents based on an explicit time model. Each time the agent is affected by some events, the simulation waits for the agent's reactions to be scheduled (Anderson 1997). However, if the simulation and the agents shall be executed by a concurrent, distributed simulation mechanism in an efficient manner the synchronization becomes more difficult (Theodoropolous and Logan 1999, Logan and Theodoropoulos 2000, Uhrmacher and Gugler 2000).

In the work by Uhrmacher and Gugler (2000) a simulator has been introduced in James which splits simulation and external deliberation into different threads. It allows simulation and deliberation to proceed concurrently by utilizing simulation events as synchronization points. The simulation proceeds only if the already consumed time of the deliberation processes exceeds the current time step. Thus, the simulator gives a guarantee that the deliberation will finish at a simulation time which lies after the "current" simulation time. Since a simulator might finish before the "current" time and other processes might have processed events being due at the proposed time, a rollback to the very last state might prove necessary. Stepwise and concurrently, the simulation and agents approach the wall-clock and simulation time at which an external process will finally complete its execution. On the way, other agents can start and finish external processes and the simulation can proceed. At a simulation time, which is calculated according to the time model, the simulator places the results of the external process into the peripheral port Z of the atomic model. At that time an internal or confluent transition will be enforced. Based on the state and the content of the peripheral port, the model produces an output and determines the next state. Originally developed to relieve the simulation from heavy deliberation processes, it turned out, that for plugging agents into simulation this kind of simulator is particularly useful.

However, we slightly re-defined the simulator to support an arbitrary number of concurrent processes associated with each atomic model and not just one as the prior version allowed. Also we changed the optimistic into a conservative simulation strategy to avoid inconsistent states of the agents which are external to the simulator. In the original algorithm, models whose normal events were due were asked to process the events and concurrently other models that had an external process running were asked for a guarantee. Now, first all models with external processes running are asked for guarantees for the current time. Only thereafter normal events which are due at the, meanwhile possibly updated, "current" time are processed.

6 CONCLUSION

Test beds provide dynamic environments to conduct controlled experimentation with agents. Thereby, they follow different approaches. Some handle agents as external processes loosely coupled to the simulation by the exchange of events. Other simulation systems describe agents within the modeling formalism. Therefore, they provide pre-defined frames based on which an agent's structure can be modeled and modules of an agent can be invoked.

By testing and plugging `Mole` agents in `James` we nurtured both approaches. During simulation agents are run as they are run in their normal execution environment. On the other hand each agent is associated with a model, a kind of representative within the virtual world, whose behavior and structure reflects discrete changes during the agent's simulated life and which serves as its individual interface to the simulation.

Simulation and agents run concurrently. Based on explicit time models defined by the user, the simulation engine of `James` supports to execute multiple agents with dynamic patterns of interaction and composition in distributed environments, and synchronizes soundly the activity of external threads with the simulation system.

Thus, for the modeler the testing of agents requires besides composing the experimental frame hardly any extra effort. However, to provide this service for certain agent classes in a general simulation system, the methods which are directed from the agent to its environment have to be re-implemented. This effort should not be under-estimated but fortunately, as the definition of the representative class, has only to be done once and will partly be re-usable in supporting other mobile agent systems. The communication from simulation to the `Mole` agent we got for free due to the extensive use of `Java` reflection.

ACKNOWLEDGMENT

We would like to thank the `Mole` research group for their cooperation and for facilitating our work tremendously by making the source code available.

REFERENCES

S.D. Anderson. 1997. Simulation of multiple time-pressured agents. In *Proc. of the 1997 Winter Simulation Conference*, Atlanta.

M.S. Atkin, D.L. Westbrook, and G.D. Cohen, P.R. ad Jorstad. 1998. AFS and HAC domain-general agent simulation and control. In *AAAI'98 Workshop Software Tools for Developing Agents*.

J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Strasser. 1997. Communication concepts for mobile agent systems. In *Mobile Agents - MA'97 Proc. of the 1st. International Workshop*. Springer.

J. Baumann, 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.

J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. 1999. Towards realistic million node internet simulations. In *Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada.

S. Hanks, M. E. Pollack, and P. R. Cohen. 1993. Benchmarks, test beds, controlled experimentation and the design of agent architectures. *AAAI*, (Winter):17–42.

Saphira software manual, Saphira version 6.1. ActivMedia Robotics LLC, Peterborough, NH.

N. Itsuki. 1995. Soccer server: A simulator for robocCup. In *JSAI AI-Symposium 95: Special Session on RoboCup*.

B. Logan and G. Theodoropoulos. 2000. Dynamic interest management in the distributed simulation of agent-based systems. In H. Sarjoughian, Cellier F.e., M.M. Marefat, and J.W. Rozenblit, editors, *2000 AI, Simulation and Planning in High Autonomy Systems*, pages 45–50.

T. A. Montgomery, J. Lee, D. J. Musliner, E. H. Durfee, D. Damouth, Y. So, and the rest of the UM-DIAG. 1992. *MICE users guide*. Dep. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI.

M.E. Pollack. 1996. Planning in dynamic environments: The DIPART system. In A. Tate, editor, *Advanced Planning Technology*. AAAI Press.

G. Theodoropoulos and B. Logan. 1999. A framework for the distributed simulation of agent-based systems. In H. Szczerbicka, editor, *European Simulation Multi Conference - ESM'99*, pages 58–65. Ghent: SCS Europe.

A.M. Uhrmacher. 2000. A system theoretic approach to constructing test beds for multi-agent systems. In F. Cellier and H. Sarjoughian, editors, *A Tapestry of Systems and AI-based Modeling & Simulation Theories and Methodologies: A Tribute to the 60th Birthday of Bernard P. Zeigler*, Lecture Notes of Computer Science. Springer.

A.M. Uhrmacher and K. Gugler. 2000. Distributed, parallel simulation of multiple, deliberative agents. In *Parallel and Distributed Simulation Conference PADS'2000*, Bologna, May. IEEE Computer Society Press.

A.M. Uhrmacher and B. Schattenberg. 1998. Agents in discrete event simulation. In *European Simulation Symposium - ESS'98*, Nottingham, October. SCS.

A.M. Uhrmacher, P. Tyschler, and D. Tyschler. 2000. Modeling and simulation of mobile agents. *Future Generation Computer Systems*.

M.J. Wooldridge and N.R. Jennings. 1998. Pitfalls of agent-oriented development. In *Proc. 2nd Internation Conference on Autonomous Agents (Agents-98)*, Minneapolis.

B.P. Zeigler, S.B. Hall, and H.S. Sarhoughian. 1999. Exploiting HLA and DEVS to promote iteroperability and reuse in Lockheeds corporate environment. *Simulation*, 73(4).

B.P. Zeigler, H. Praehofer, and Kim T.G. 2000. *Theory of Modeling and Simulation*. Academic Press.

AUTHOR BIOGRAPHIES

ADELINDE M. UHRMACHER is an associate professor in the Department of Computer Science at the University Rostock. Her research interests are artificial intelligence, modeling and simulation, particularly the development of agent-oriented modeling and simulation methods. Her e-mail and web-page addresses are `<lin@informatik.uni-rostock.de>` and `<www.informatik.uni-rostock.de/~lin>`.

BERND G. KULLICK is finishing his master's degree in the Faculty of Computer Science at the University Ulm. His major research interests are mobile and intelligent agents and their evaluation. His e-mail address is `<bk1@informatik.uni-ulm.de>`