Reflection in Simulating and Testing Agents

A.M. Uhrmacher & Bernd Kullick & Jens Lemcke
Department of Computer Science, University of Rostock, D-18051 Rostock, Germany
lin@informatik.uni-rostock.de

Abstract

Simulation methods offer an experimental approach to analyzing the dynamic behavior of multi-agent systems. If agents are specified in the modeling language and become part of the simulation, the simulation system has to support reflection, i.e. models which assess and access their own structure and behavior. In JAMES, a Java-based agent modeling and simulation environment, models of dynamic systems including agents are described as composite, time triggered, and reflective state automata that are able to change their own structure and behavior. Often agents are tested by interacting with the simulation environment as a kind of black box. To support experimenting with agent models and agents, equally, in JAMES models can serve as interfaces. During executing the agent, these models reflect the agent's behavior projecting it into the simulation. This type of interaction is facilitated, as will be shown, with the mobile agent system MOLE, if the agent system is implemented in JAVA, because in this case the mechanisms of reflection that JAVA offers can be utilized.

1 Introduction

The more complex applications of agents become in terms of number of agents, nodes to be visited, or deliberation capabilities, the more the effort will pay to thoroughly specify agents and analyze the performance of their different planning, cooperation or moving strategies in virtual environments. As agents are aimed to function in open dynamic environments, simulation methods have been employed for analyzing the agent's behavior [7; 10; 9; 4; 5].

The ability of agents to adapt their own interaction, composition and behavior pattern challenges the expressiveness of modeling and simulation formalisms (e.g. [2; 8; 17]). To support the modeling of agents the simulation and its underlying formalism have to support reflection, i.e. models which assess and access their own structure and behavior. Among the formal approaches to discrete event simulation the DEVS formalism has been used as a basis for exploring possibilities and implications of integrating and expressing variable structures since the 80ties [20]. Whereas most discrete event simulations pertain to events and processes as elementary units, DEVS describes dynamic systems as time and event triggered state automata, a perception which coincides with an acknowledged interpretation of agents. JAMES, a Java-Based Agent Modeling and Simulation Environment, adds reflection to the DEVS formalism to capture the notion of self aware and self manipulating agents.

The implementation and application of dynamic test scenarios for multi-agent systems requires considerable modeling effort. Typically, the agent is not modeled in its entirety. E.g., one of the first simulation systems for agents allowed to plug code fragments, or single modules into the skeleton of an agent model [11]. To reduce the modeling effort agents are sometimes treated as external source and drain of events [13; 1] with the simulation being interpreted as black box. The loose coupling of simulation and agents saves the user the extra effort to specify the agent in the modeling language of the simulation system. However, typically more effort is required to analyze the interaction and actions of agents in the virtual world. Agents are not explicitly represented in the test environment and their behavior can only be analyzed based on the induced effects.

On the one hand, one would like to have the possibility to execute agents as they are switching arbitrarily between an execution in the real environment and in the virtual test environment [19; 14]. On the other hand, one would like to have agents specified as integral part of the experimental setting and as such perceivable and controllable. It would help to focus the view on relevant aspects and changes within the agents.

To meet both necessities the idea of representatives is introduced in JAMES [19]. A discrete model is associated with the actual agent which represents the agent and reflects the behavior of the agent which runs concurrently to the simulation. The potential of this idea we will explore with the mobile agent system MOLE [3] which is implemented in JAVA. The use of JAVA reflection methods facilitates plugging MOLE

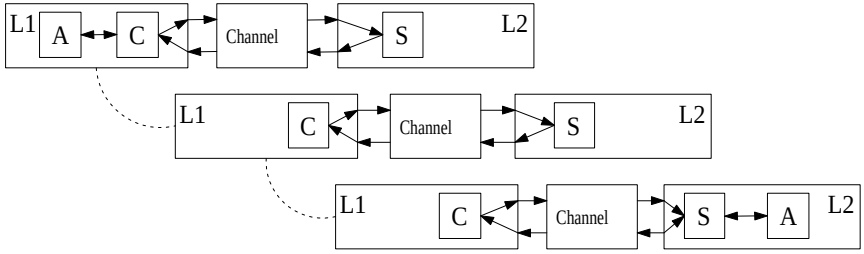

Figure 1: An agent model moving between two locations

agents into JAMES and introduces the third kind of reflection we wish to explore in simulating and testing agents.

2 Composite, reflective time triggered automata

JAMES is based on DEVS [20] which belongs to the formal and general approaches to discrete event simulation. The model design is coined by a hierarchical, compositional construction of models. It distinguishes between atomic and coupled models. The role of the coupled model is to define the interaction structure between its components. Coupled models have no behavior of their own. The behavior of an atomic models is defined by transition functions, an output function, and a time advance function which determines how long a state persist "per se". An internal transition function dictates state transitions due to internal events, the time of which is determined by the time advance function. The external transition function is triggered by the arrival of external inputs. This allows the modeler to describe agents and their environment as time-triggered composite automata.

As do other formal approaches to discrete event simulation, DEVS presupposes static model structures and does not provide means for reflection; both are crucial in modeling and simulating agents. To support models that are able to assess and access their composition, interaction, and behavior structure from an agent's perspective, a formalism has been developed which expresses agents as reflective, time-triggered automata [17] (Figure 2). As the coupled model holds the information about composition and interaction between components, a change of composition or interaction, even though induced by an atomic model, takes effect at the level of the coupled model. The formalism shall not be described in more detail in this paper (see [17]).

Executing the model according to the user's specification and the given initial situation is the task of a discrete event simulator. Each model is interpreted and executed by a tree of processors which reflects the hierarchical compositional structure of the model. Each of the processors is associated with a component of the model and 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.

Based on this processor tree different distributed, parallel execution strategies have been implemented in JAMES. Whereas one adopts a conservative strategy where only events which occur at exactly the same simulation time 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. The performance of the different strategies to test planning agents are compared in [18].

2.1 An Example - Modeling Mobility

Modeling the movement of agents in JAMES as an example for variable structure models requires the utilization of the different transition functions, and the output function of the atomic model [16]. The process of moving comprises adding and removing model components from coupled models (Figure 1), modifying the interaction structure within the coupled models, and, in addition, the possibility of sending references, i.e., names or model components within messages.

In the scenario depicted in Figure 1 the model component that represents a client `C` requests a task from another model component that represents an agent `A`. The agent model responds by invoking its external transition function and decides to move to the location `L2`. Both locations `L1, L2` are represented by coupled models. After some simulation time has elapsed as determined by the time advance function, the agent model charges its port with a request to migrate and thus initiates the migration. Output function and internal transition function are intrinsically connected; they form a unity and are invoked at the same simulation time. This offers the opportunity to update the state of the agent model at the moment at which the migration starts. The model ceases to exist within the former location `L1`. The structure of the coupled model `L1` has changed. The time the movement will take to complete depends on the model which is located on the path towards the destination, i.e. `Channel`. The `Channel` that connects both locations `L1` and `L2` might be modeled as a simple atomic model, a coupled model or even represent an entire network simulation system.

Finally, the message including the agent `A` will reach

$dynDEVS =_{df}$ $\langle X, Y, Z, m_{init}, \mathcal{M}(m_{init})\rangle$ with
X, Y sets of model inputs, model outputs,
Z_i, Z_o inputs and outputs from and to external processes
$m_{init} \in \mathcal{M}(m_{init})$ the initial model
$\mathcal{M}(m_{init})$ is the least set having the structure
$\{\langle S, s_{init}, \delta_{ext}, \delta_{int}, \rho_\alpha, \lambda, ta\rangle |$
S set of states
$s_{init} \in S$ the initial state
$\delta_{ext} : Q \times X \to S \times Z_o$ the external transition function is triggered by the arrival of events which have been produced by other models (its influencers) with $Q = \{(s, z, e) : s \in S, z \in Z_i, 0 \leq e \leq ta(s)\}$
$\delta_{int} : S \times Z_i \to S \times Z_o$ the internal transition function is triggered by the flow of time
$\rho_\alpha : S \times Z_i \to \mathcal{M}(m_{init})$ the model transition function determines the next "incarnation" of the model in terms of state space and behavior pattern
$\lambda : S \times Z_i \to Y$ the output function fills the output port and will trigger external transitions within influenced components
$ta : S \times Z_i \to \mathcal{R}_0^+ \cup \{\infty\}$ the time advance function determines how long a state persists
$rc2st : S \times Z_i \to \mathcal{R}_0^+ \cup \{\infty\}$ the time model relates resource consumption of external processes to the simulation time $\}$

and satisfying the property

$$\forall n \in \mathcal{M}(m_{init}).(\exists m \in \mathcal{M}(m_{init}).\ n = \rho_\alpha(s^m)\ with\ s^m\ \in S^m) \vee n = m_{init}$$

Figure 2: The formalism behind JAMES that allows models to change their own state and behavior pattern and to interact with other models and with external processes.

its destination S. The receiver S will be activated via its external transition function and will be asked to insert the agent model into the new location L2. After inserting the agent into its new location, S will wake up the moved agent by sending a welcome message. At that moment the external transition function of the agent model serves as an entry point to resume processing according to the input, code, and state of the agent.

In the above paragraphs we sketched a single simulation run based on modeling agents and their environment entirely in JAMES. Since both, i.e. mobile agents and their environment, have been modeled the modeling formalism served as a specification language. In the example the agent model only represents a mobile agent but has not been connected to an agent or a part of agent code running.

2.2 Interface to External Processes

In the above example a communication with external processes does not take place. Agents and their virtual environment are entirely modeled in JAMES. To test planning and commitment strategies of agents [15], JAMES has been equipped with outer ports, which are now used to support the interaction of atomic models with external processes in general.

The classical ports of DEVS models collect and offer events that are produced by models. The outer ports in JAMES allow models to communicate with processes that are external to the simulation. Thereby, not the entire simulation system as one black box interacts with external agents, but each single model can function as an interface to external processes. If agents and simulation shall interact in simulation time, a function transforms external resource consumption into simulation time (Figure 2). All functions, including state transition functions, model transition function, output function, and time advance function are also based on, and partly directed to the outer ports. The external process fills the outer ports at a time when the external process finishes its execution or at a simulation time which is calculated by applying the time model to the resource consumption. The model offers its events to the external process via the outer ports (see Figure 2). This mechanism can not only be used to invoke external planners from inside the modeled agent but to connect a model to an agent as a whole which runs concurrently to the simulation.

3 Representatives - "Reflecting" An Agent's Behavior

Typically, the agent is not modeled in its entirety, as in the above simple example, but part of the agent is modeled and part of the agent's modules, e.g. planners are invoked. To reduce the modeling effort agents are often treated as external source and drain of events. Requests and messages that are normally sent to the agent's real environment are redirected to the simulation system. Agents and simulation system are synchronized in simulation time, in this case messages exchanged between agents and simulation are labelled with time-stamps [13; 1], or they interact in wall-clock time. An example

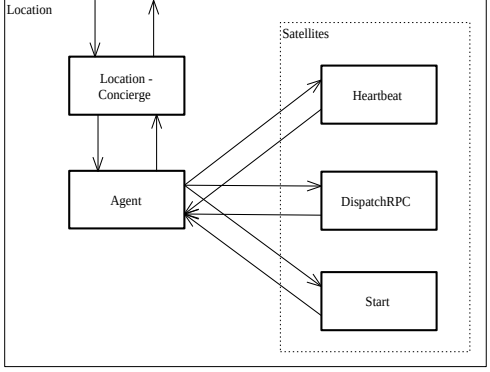

Figure 3: Locations, agents and their processes in JAMES

for the latter is the soccer simulator of the RoboCup initiative [12]. It checks frequently whether any agent has produced a message for the simulator, otherwise the simulator proceeds simulating. Slowing down the execution of the simulation engine diminishes the time pressure for the agents. The purpose of these type of simulation is to support competition games rather than a thorough testing which requires more control of the experiment. Recently the introduced bias has been analyzed to improve the synchronization between simulator and agents [6].

A loose coupling allows to switch arbitrarily between an execution in the real environment and in the virtual test environment. By modeling agents they become an integral part of the experimental setting. The idea of representatives is to allow models to be associated with an actual agent to combine the benefits of both approaches. Atomic models not only represent the agent but reflect, i.e. give evidence of, the actual agent's behavior.

In the following we will define representatives for agents of the mobile agent system MOLE.

3.1 Mole

MOLE represents a Java-based mobile agent system [3]. Engines, which represent the MOLE runtime system, transform and forward messages between 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. 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. Agents can comprise a dynamic set of concurrent running or waiting threads and are not restricted to one line of activity.

3.2 The Representative

The life of an agent starts in the moment a location initiates the creation of an agent. To become an active member of an agent's society the preparation

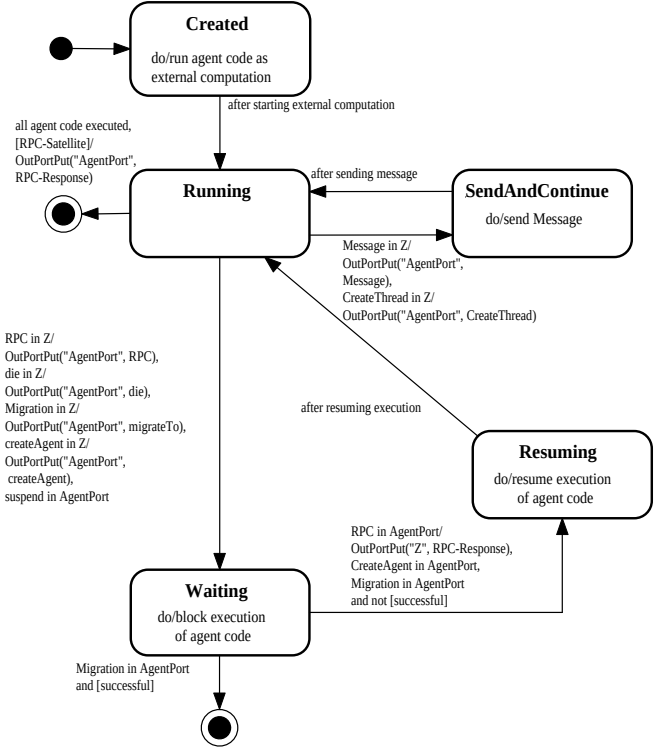

Figure 4: The JAMES model of a satellite described as a statechart

method signs responsible. The preparation method is invoked at the time at which an agent is created or just awakened after a successful migration. Thereafter, the working phase of an agent starts, which includes activating the start method and the heart beat (a method which is invoked with a certain frequency), 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. In JAMES a MOLE agent is represented as one model surrounded by models that represent its running or waiting threads (Figure 3).

At the moment, e.g. a RPC reaches the agent core model, the core agent will create a satellite to dispatch the remote procedure call (Figure 4). The satellite's states, i.e. `Created`, `Running`, `SendAndContinue`, `Resuming`, and `Waiting`, reflect the phases that the thread undergoes during execution.

The satellite will transform the incoming request by using the JAVA reflection into calling a concrete method of the MOLE agent. The satellite itself changes to the state `Running`. In the opposite direction significant events of the executing MOLE thread, e.g. the invocation of the migration method, are translated into events, directed to the simulation system which the satellite will forward to the agent core model (Figure 5). In the case of a migration request, the core agent will ask all its satellites to suspend current

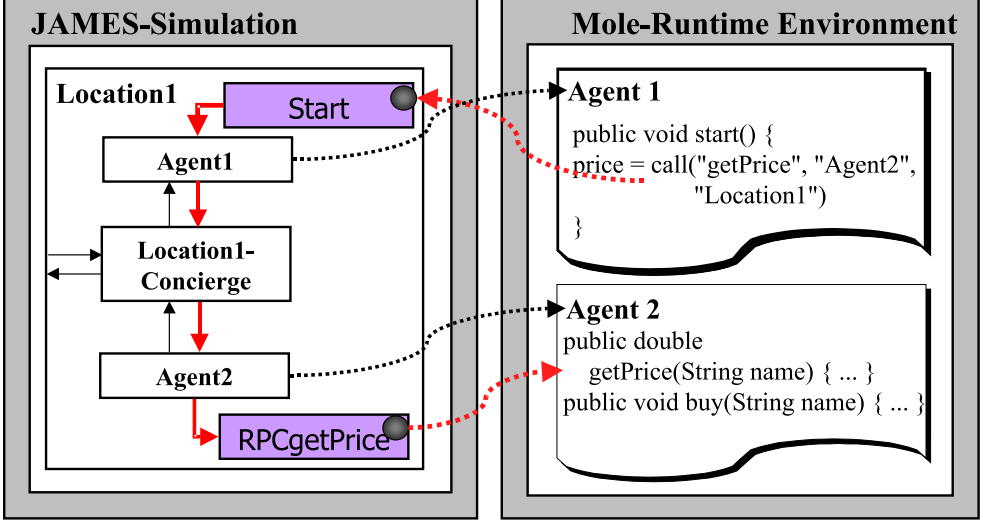

Figure 5: Intertwining MOLE and JAMES

threads, launches the migration request into the network, and will turn into the state `migrating`. During this time the agent only represents a shadow of itself. It waits for either an acknowledgement of a successful migration or a notification that the migration has failed. In the former case, the agent has been successfully installed at its new location and has invoked its start routine, at the old location the shadow is no longer needed. It informs all satellites to stop themselves and commits suicide. In the latter case something has hindered the installation of the agent in its new location, in this case the shadow becomes lively again, and informs its satellites to resume their activities.

Some of the state changes are initiated by incoming simulation events produced by other models. Other state changes of the model are initiated by the associated agent.

4 Java Reflection Intertwining Mole and James

Methods in MOLE are not simply executed as JAVA methods but reflected to make sure that the execution adheres to the security policy. Due to this mechanism of reflection within MOLE, the invocation of methods can easily be identified and the calling thread can be suspended to be resumed afterwards. Methods of the MOLE API which constitute the interface between MOLE agents and their run-time environment, have only to be slightly changed to redirect calls and messages to the simulation system. Those methods fill in automatically an outer port and trigger the state changes of the atomic model. The port is filled at a simulation time which is determined by applying a function to translate the ressources consumed into simulation time. The ressources have been consumed inbetween starting the thread and the thread reaching a method invokation that is directed to the environment of the agent. Executing one of these methods results in charging an outer port of the associated satellite, i.e. `Z` (Figure 4), and in suspending the executing thread of the MOLE agent. Later the thread will be resumed or stopped by the satellite model. Thus, each agent to agent communication in MOLE is transformed into a communication from MOLE agent to JAMES simulation and back. In the opposite direction, if events produced by models reach the agent core model, the core agent model will either create new satellites which will generate a new agent thread by invoking a method via JAVA reflection, or will forward the message to an existing satellite which will resume the suspended thread (Figure 5).

Whereas the agent core model represents the central focus of control, its satellites provide the interfaces to the agent's processes. All of them implement an abstract view of the state and the behavior of an agent. An agent's behavior is described by piecewise constant trajectories, where each episode is separated from the next by the occurrence of a discrete event, e.g. the sending or receiving of messages and calls. Not only MOLE agents but also MOLE locations and engines are associated with representatives, which themselves are part of a network simulation. Messages, calls, and agents are propagated through the virtual network based on the simulation mechanisms provided in JAMES, and according to the actual model of the physical network which underlies the experiment.

5 Conclusion

Our approach shows how a general modeling and concurrent simulation formalism for multi-agent systems, i.e. JAMES, can be adapted to test agents of a concrete mobile agent system and how different forms of reflections come in handy. By using the JAVA reflection and slight changes of the MOLE API, MOLE agents can be plugged into the models without changing the source code. Agents are thus associated with a model which

serves as their representative during the simulation and reflects crucial state changes and events during the agent's lifespan including the adaptation to environmental changes and new requirements. For that purpose, JAMES supports the definition and execution of reflective models with the ability to represent, control, and modify their own behavior. By combining these types of reflection the representative can also be used in early phases to test single strategies and agent components in isolation. Thus, the agent developer gains flexibility how and when to test his or her agents, whether as abstract model, as frame partly filled with software components or in its entirety. The programmer can switch arbitrarily between an execution in the real environment and the virtual test environment. The interaction of agents and simulations via models facilitates systematic experiments to analyze the behavior of multi-agent systems. However, so far only few experiments with small networks have been executed to demonstrate the feasibility of the approach.

References

[1] S.D. Anderson. Simulation of Multiple Time-Pressured Agents. In *Proc. of the Wintersimulation Conference, WSC'97*, Atlanta, 1997.

[2] A. Asperti and N. Busi. Mobile Petri Nets. Technical Report UBLCS-96-10, University of Bologna, 1996.

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

[4] L. Bernardo and P. Pinto. Scalable Service Deployment using Mobile Agents. In K. Rothermel and F. Hohl, editors, *2nd Workshop on Mobile Agents 98 (MA98)*, volume 1477 of *Lecture Notes on Computer Science*, pages 261–272. Springer, 1998.

[5] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile Agents in Distributed Information Retrieval. In M. Klusch and K. Sycara, editors, *Intelligent Information Agents*. Springer, 1999.

[6] M. Butler, M. Prokopenko, and T. Howard. Flexible synchronisation within robocup environment: A comparative analysis. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer: World Cup IV*, number 2019 in LNAI, pages 119–128, London, 2001. Springer.

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

[8] M. Köhler, D. Moldt, and H. Rölke. Modelling the structure and behavior of petri net agents. In J.M. Colom and M. Koutny, editors, *Applications and Theory of Petri Nets 2001*, number 2075 in LNCS, pages 224–241, London, 2001. Springer.

[9] G. Di Marzo, M. Muhugusa, and C.F. Tschudin. A Survey of Theories for Mobile Agents. *World Wide Web Journal*, pages 139–153, 1998.

[10] N. Minar, K. Hultman-Kramer, and P. Maes. Cooperating Mobile Agents for Mapping Networks. In *First Hungarian National Conference on Agent-Based Computing*. John von Neumann Computer Society, 1998.

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

[12] I. Noda. Soccer Server: A simulator for Robo Cup. In *JSAI AI-Symposium 95: Special Session on RoboCup*, 1995.

[13] M.E. Pollack. Planning in Dynamic Environments: The DIPART System. In A. Tate, editor, *Advanced Planning Technology*. AAAI, 1996.

[14] H. Sarjoughian, B.P. Zeigler, and S.B. Hall. A Layered Modeling and Simulation Architecture for Agent-Based System Development. *Proceedings of the IEEE*, 89(2):201–213, 2001.

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

[16] A. M. Uhrmacher, P. Tyschler, and D. Tyschler. Modeling Mobile Agents. *Future Generation Computer System*, 17:107–118, 2000.

[17] A.M. Uhrmacher. Dynamic Structures in Modeling and Simulation - a Reflective Approach. *ACM Transactions on Modeling and Simulation*, to appear.

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

[19] A.M. Uhrmacher and B. Kullick. Plug and Test Software Agents in Virtual Environments. In *Winter Simulation Conference - WSC'2000*, Orlando, FL, December 2000.

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