Processing dynamic PDEVS models

Jan Himmelspach, Adelinde M. Uhrmacher
Department of Computer Science and Electrical Engineering
Modeling and Simulation group
Albert-Einstein-Str. 21, 18059 Rostock, Germany
jan.himmelspach/adelinde.uhrmacher@informatik.uni-rostock.de

Abstract

Structural changes, i.e. the creation and deletion of components, and the change of interactions are salient features of adaptive systems. To model and specify these systems variable structure models are required, i.e. models that entail in their own description the possibility to change their structure. To execute these models a simulator with a clear semantic of intertwining structural and non-structural changes is required.

In JAMES *different simulator components, e.g. for paced, unpaced, sequential, and parallel simulation, support the continuous use of models and simulation from specification to testing and a composition of the simulation engine on demand. Two types of simulator components for variable structure models are developed, integrated into the simulation layer, and the implications discussed.*

1. Introduction

Variable structure models play traditionally an important role in areas like ecology and biology, processes like succession of ecological systems and differentiation at cellular level are prominent representatives of these classes [26, 4]. The more software systems shall work in open dynamic environments or exhibit properties like autonomy, flexibility, self-organization, or regulation, the more structural phenomena like the emergence of new organization structures, the generation and the loss of components become of interest. Examples are providing services in AD HOC networks [9], or the new initiative directed toward building regenerative software systems [14]. Although systematic experimentation is not as often used in Computer Science as in other scientific areas [22], the need for modeling and simulation as a tool to support the design and analysis of software systems increases with the number of concurrent, distributed software systems that shall run in open, dynamic environments. In the area of multi-agent systems, modeling and simulation has found its place to explore abstract phenomena of cooperation and coordination based on modeled agents [12], for testing agent implementations in competitive scenarios like ROBOCUP and ROBOCUPRESCUE, [21], and for evaluating the performance of agent systems [19]. In the context of embedded systems models of the software [18] and models of the environment [16] are used to automatically generate test cases. Generally, the continuity of modeling and simulation throughout software development processes is suggested [11] and first challenges for establishing modeling and simulation in the software design cycle have been identified [23]. Models that specify the behavior of software systems are commonly used in designing software, e.g. STATECHARTS specify the adaptive behavior of agents and their roles, e.g. [31, 15, 5], as do Petri Nets, e.g. [32, 13]. These models are used to automatically generate source code, e.g. [15], to analyze certain properties of the system [32], or for simulation, e.g. [13]. A clear operational semantics which defines the simulation of the modeled systems is seen as an asset of any modeling formalism. E.g. the operational semantic of the stochastic pi-calculus has been implemented in a discrete event simulator. However, the operational semantics of models, which is used for simulation, has seldom been as emphasized as in DEVS. When Zeigler developed the modeling formalism in the 70s and 80s, he specified the operational semantics in the abstract simulator and all DEVS variants have felt obliged to do so afterward [30]. This clear statement how a DEVS model is executed has been seen as one of the advantages of using DEVS [3, 34]. In the following we define abstract simulators that allow to execute variable structure models that are based on the PDEVS (ParallelDEVS) [33] formalism in a flexible and unambiguous way.

2. Background

The model design in DEVS distinguishes between *atomic* and *coupled models*. Coupled models are the means to develop complex models by hierarchical composition. A

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 IEEE

coupled model is a model consisting of different components and specifying the coupling of its components. Its interface to its environment is given by a set of external input and output events. The description of an atomic model embraces a set of input events, a state set, a set of output events, an internal, external, and confluent transition function, an output and time advance function. The internal transition function (`deltaInt`) dictates state transitions due to internal events, the time of which is determined by the time advance function. The external transition function (`deltaExt`) is triggered by external inputs, which are defined as bags over simultaneously arriving input events. The confluent transition function (`deltaCon`) is invoked if internal and external events coincide. The abstract simulator is built by a tree of processors. The leafs are simulators that are responsible for processing atomic models. Internal nodes are the coordinators, which propagate activation and input output messages through the processor tree. The original DEVS formalism [33] does not support the modeling of dynamic structures. It has been extended to describe model components and couplings being created, removed, e.g. [1, 27, 33]. Our modeling and simulation system James (A Java-Based Agent Modeling Environment for Simulation) is based on the DYNDEVS formalism [27], as it supports variable structure models by means for reflection: a model changes its own structure. Atomic models are responsible for inducing structural changes by model and network transitions. No dedicated controller resides over them as required in other approaches, e.g. [1]. JAMES forms a component-based modeling and simulation environment for multi-agent systems. The component based design idea permeates the entire environment: same as models can be composed of model components, the simulation engine can be composed of simulator components on demand [10]. Simulator components for a paced, unpaced, sequential and parallel simulation have been implemented. Based on the components the simulation engine can be adapted to the model's needs, hardware resources, and the user's preferences. Whereas a simulation engine exists that supports variable structure [29], this simulation engine is not component-based and puts some restrictions on the execution of variable structures, i.e. an atomic model can only delete itself, change its own interaction structure and add new models in the coupled model it resides in, for all other structural changes a model has to negotiate. These restrictions were adopted to guarantee the autonomy of agents. However, these restrictions are not necessary for designing a simulation engine. In addition, the metaphor of self-determined agents burdens the modeling of variable structure systems that embrace multiple reactive entities accessing each others structure unnecessarily. Thus, to leave the freedom of choice which metaphor to adopt to the modeler, the new simulation engine puts little constraints on the initiation of structural changes.

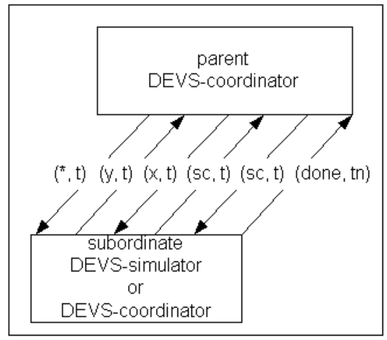

Figure 1. dynPDEVS simulator protocol

3. Parallel execution of variable structures in DEVS

Following the argumentation line of DYNDEVS, structural change requests can be issued by any atomic model during one of its transition functions. Although the atomic model invokes the structural changes during the transition function, their actual execution is delayed. Some of the induced structural changes can be realized by the model itself, others will be sent up the model hierarchy. The structural change requests are buffered and executed after all models have finished their state transitions for the actual time t, but before the timeAdvance method of the models is executed. Only if the structural changes at one level have been completed and the done messages have been received, structural changes at the next higher level of the simulator and model tree are executed.

Since a model can induce changes at any place in the model tree an additional complete message passing process, comparable to the x and y message handling in the DEVS abstract simulator, needs to be introduced. This means that a model can be activated by either a *, x, or sc (structure change) message. Structural changes are always executed from bottom to top, i.e. first structural changes at the level of atomic models are executed (ρ_α function in the DYNDEVS definition), followed by the structural changes to be applied at the level of the coupled model. Which messages are propagated through the simulator tree in which direction can be seen in Figure 1. Each arrow from the parent to a child forms an entry point into the protocol which is executed from left to right. This necessitates that null messages are sent in case no actual inputs, cp. [33], or structural changes have to be processed.

3.1. An abstract simulator

For ease of reading we have not split up the code according to our `pre-`, `do-`, and `postEvent` template.

```
01  send * to child
02  wait for y message from child
03  send x message to child
04  wait for sc message from child
05  send sc message to child
06  wait for done message from child
```

Figure 2. Pseudo Code of the next step method of the root coordinator

The RootCoordinator in Figure 2 receives an incoming structure change message (04) and sends an (empty) structure change message to its child (05). A received structure change message is currently not executed - because in the current version it is assumed that a coupled model embraces the entire model and all activities, structural and non structural, happen within it. The remaining lines of the root coordinator's code describe the usual DEVS processing: I.e. sending the * message (01), waiting for the y message (02), sending the empty x message (03), and finally waiting for the done message (06).

```
01  when receive *, x or sc message
02    if message is * or x message
03      if message is * message
03        receive * message
04        send * to imminent children
05        wait for y messages from those children
06        send y message to parent
07        wait for x message from parent
08      fi
09      receive x message
10      send x messages to all influenced and
        imminent children
11      wait for sc requests
12      send structural change message to parent
13    fi
14    receive sc message from parent
15    send sc message to children
16    wait for done messages from any
      activated children
17    execute sc requests
18    send done message
19  end when
```

Figure 3. Pseudo Code of the dynamicPDEVS coordinator

The coordinator in Figure 3 is activated either by an incoming *, x, or sc (structure change) message. The coordinator has now to handle incoming sc messages from its parent and its children, which implies waiting for them (11,14), and propagating them (12,15), and executing those that refer to the associated coupled model of the coordinator (17). Thus, all structural changes that affect models located below the coupled model are executed before the corresponding coordinator sends its done message (18) to its parent.

```
01  when receive *, x or sc message
02    if message is * or x message
03      if message is * message
04        receive * message
05        lambda
06        send y message to parent
07        wait for receive x message
08        if isEmpty (x message)
09          deltaInt
10        else
11          receive x message
12          deltaCon
13        fi
14      else
15        receive x message
16        deltaExt
17      fi
18      send sc message
19    fi
20    receive sc message
21    execute sc requests
22    send done message
23  end when
```

Figure 4. Pseudo Code of the dynamicPDEVS simulator

The simulator in Figure 4 can be activated (as the above Coordinator) by a *, x, or sc message. A * or a x message starts the normal DEVS processing. To complete the processing a message with all structural changes buffered during the δ-functions (or null if none) is sent to the parent (18). Afterward the simulator waits for a structure change message from its parent, which might be null, though (20). At line 20 the simulator will start its computation if activated by a sc message. If structural changes are due they will be executed (21). Finally, the simulator sends a done message to the parent (22).

3.2. Coordinating structural change messages

Structure change messages are executed bottom up the simulator hierarchy. Each simulator and coordinator filters the messages that it will process itself. At each coordinator the messages requesting structural changes are processed in the following order: 1. Create models, 2. Create couplings, 3. Remove Couplings, 4. Remove models

1 before 2: Couplings can only be added to existing models. Therefore it is important to guarantee that all models are created before the couplings are added. Otherwise the adding of couplings might fail.

3 before 4: If a model is removed all its couplings are removed as well. If a coupling that does not exist shall be removed this might be interpreted as an error.

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

1 before 4: A removal of a model is valued higher than invoking the creation of the same model.

2 before 3: A removal of a coupling is valued higher than invoking the creation of the same coupling.

The latter two rules prevent nondeterministic model behavior: If two models want to change structures in a way that one of the models adds entity (model or coupling) x while the other model removes x the execution could either end up with an exception (added an already existing entity) or simply with x existent. However, the result would depend on the (random) order of the requests. Please note that structural changes are induced by atomic models which do not have access to the actual overall model structure, they rely on their knowledge about the model structure, which might be wrong.

Structural change requests that are not executable, are (a) the entity to be removed is not there, (b) the entity to be added is already there or (c) a coupling shall be created with at least one not existing model.

By default the system will throw an exception which immediately stops the simulation. However, as there may be scenarios where this shall not been interpreted as an error, these exception mechanism can be turned off: neither the change inducing model nor the modeler becomes aware of the fact that a change has failed.

The mapping of the structural change requests to the actual structure is based on knowing the names of models. If a model wants to add another model to the coupled model, it belongs to, or wants to delete one, it only needs its short name. If a model wants to access the components of another coupled model it has to know the full name, which is built by including all the coupled models the accessed component is nested in. Thus, the modeler can restrict the possibility to induce structural changes directly by making a model only aware about the names of the models that belong to the same coupled model, to realize the old JAMES strategy. To mimic the controller suggested by Barros [2], an atomic model could be defined as a kind of concierge for each coupled model, which knows about all model-components and their interactions in the coupled model it resides in and which is informed about all changes.

4. Adding external processes

EPI (external process interface) processing means that a model has an interface to an external process through which model and external processes can communicate while the simulation continues. Therefore, the definition of atomic models is extended by peripheral input and output ports. The peripheral ports can be interpreted as part of the state, as they are accessed each time the state is accessed: all functions read the peripheral ports and the transition functions are responsible for filling the peripheral output ports.

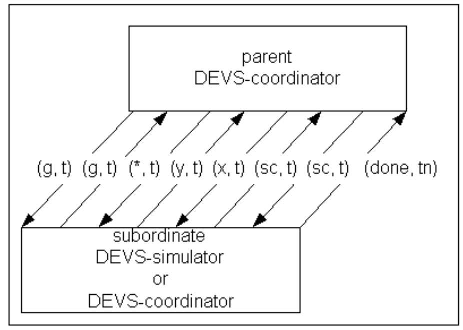

Figure 5. DEVS simulator protocol with guarantees

The simulation layer is responsible for synchronizing simulation and externally running software. Simulation time and wall clock time can be used for synchronizing, in the latter case the simulation should run in paced mode - relating simulation progress to the progress of wall clock time [7]. In the following we will focus on the unpaced variant and a synchronization in simulation time. If the externally running software does not provide information about simulation time, the time model which is associated with an atomic model can be used for that purposes. It maps the resource consumption of the external process into simulation time.

Synchronization in the unpaced parallel variant is done by using a guarantee asking mechanism and explicit time models for each external process [24]. Before a (*, t) message for any t is sent all simulators are asked for a guarantee that the next input (from the external process) to the model does not arrive before the time that shall be processed. The guarantee message precedes the star message (Figure 5). Unlike the passing of *, x/y, and sc messages, guarantee messages are processed in a separate pulse from top to bottom and bottom to top of the simulator hierarchy. As long as external processes are running the guarantee pulse alternates with the original simulation pulse, which is responsible for processing structural and non structural events.

4.1. An abstract simulator

The RootCoordinator in Figure 6 differs from the one introduced in Figure 2. First a dynamic epi PDEVS RootCoordinator asks for a guarantee for the next tn (time of next event), to ensure that no external process will deliver any result with a smaller time stamp than tn. If one of the guarantee requests returns a guarantee for a time less than tn instead of the guarantee for time tn then this value will represent the new time of next event that will be processed

```
01  send guarantee request for tn
02  wait for receive guarantee answer
03  if guarantee time < tn then
04    tn = guarantee time
05  send * to child
06  wait for receive y message from child
07  send x message to child
08  wait for receive sc message from child
09  send sc message to child
10  wait for receive done message from child
```

Figure 6. Pseudo Code of the dynamic epi PDEVS root coordinator

in the simulation (04). The remainder of the RootCoordinator processing is equivalent to the code of the previously introduced one.

```
01  when receive *, x, sc or guarantee request
    message
02    if message is guarantee request
03      send guarantee requests to epi children
04      wait for receive guarantee affirmation
        from those
05      send min guarantee affirmation to parent
06    else
07      if message is * or x message
08        if message is * message
09          receive * message
10          send * to imment children
11          wait for y messages from those children
12          send y message to parent
13          wait for x message from parent
14        fi
15        receive x message
16        send x messages to all influenced and
          imminent children
17        wait for sc requests
18        send structural change message to parent
19      fi
20      receive sc message from parent
21      send sc message to children
22      wait for done messages from any
        activated children
23      execute sc requests
24      send done message
25  end when
```

Figure 7. Pseudo Code of the epi dynamic PDEVS coordinator

The Coordinator in Figure 7 is compared to the Coordinator in Figure 3 slightly extended. Handling guarantee messages is done like in the other parallel, unpaced, epi coordinator, see [10]. If the message received is a guarantee request this request is forwarded to all subtrees which have at least one model with an attached external process (03). Afterwards the coordinator has to wait until it receives the corresponding answers (04). As a last step in guarantee message handling the minimal guaranteed time of all received guarantees is sent to the parent processor. If the message received is not a guarantee message the same message processing as in the Coordinator (Figure 3) is started.

```
01  when receive *, x, sc or guarantee request
    message
02    if message is guarantee request message
03      wait until all time models of all external
        processes are ≥ request or finished
04      send affirmation (min time of all time
        models)
05    else
06      if message is * or x message
12          if pending ext.process msgs with
            current time
13            charge the corresponding state ports
07        if message is * message
08          receive * message
09          lambda
10          send y message to parent
11          wait for receive x message
14          fi
15          if isEmpty (x message)
16            deltaInt
17          else
18            receive x message
19            deltaCon
20          fi
21        else
22          receive x message
23          deltaExt
24        fi
25        send sc message
26        create external processes and/or
          transmit peripheral outport start value
          assignments
27      fi
28      receive sc message
29      execute sc requests
30      send done message
31  end when
```

Figure 8. Pseudo Code of the dynamic epi PDEVS simulator

In comparison with the simulator (Figure 4) an EPI simulator (Figure 8) requires to handle guarantee messages and to charge the peripheral input ports and output ports so the information from and to the external processes can be accessed. If activated by a guarantee message (02), the simulator will check all timeModels of all attached processes whether they can give the guarantee or whether the processes have finished with a smaller timeStamp (3), afterwards it sends its minimum guarantee to the parent processor. This is the typical behavior of all unpaced, parallel EPI simulators and implemented in the pre-event template method, which all of them share. If a star or external message is received and there are pending messages from an external source for the current time the peripheral input ports shall be charged (12+13). After sending sc messages to the parent and before executing structural changes at the level of the simulator, the simulator creates all external processes to be created parameterized by the values in the peripheral output ports.

4.2. The problem of migration

At the moment a model is deleted, all external processes will be stopped. If a model is created an external process might be created together with the new model. A change of coupling does not affect the externally running process. A model might migrate by changing successively its couplings and even the coupled model it resides in. In this case, the model continues to exist and the external process will not be affected by the migration. If the migration of a model means that a model ceases to exist as a model and is transported through the model as a simple message, then the question arises what shall be done with the externally running processes. If only a process is moved without the state, everything is newly created at the target domain. If the migration means that an agent migrates including its state, different migration strategies can be distinguished:

- weak: the agent and its external processes are stopped and the agent is started by using an explicit entry point or
- strong: the agent, its state and its external processes are suspended and resumed to work at the target.

The easiest solution for agents and models alike is to use the weak migration concept which implies that all externally running processes will be suspended, and in case of a successful migration will be stopped.

5. Combining Processors

The processor architecture allows the flexible combination of different processors (see Figure 9) [10] to compute a model in the most efficient way. Sequential and parallel variants can be replaced more or less arbitrarily. However, simulators that have an external process running require that all coordinators up to the root are able to handle the guarantee messages. As structural changes can not be restricted to one area of the model, all simulators should be able to handle variable structure models. So one variable structure model implies that all simulators and coordinators are able to process variable structures. So the entire tree is built from coordinators and simulators implementing the protocol for processing structural changes.

Moving, adding, or removing models introduces high dynamics into a model tree. This leads to the requirement that the processors must be adapted to the tree after each

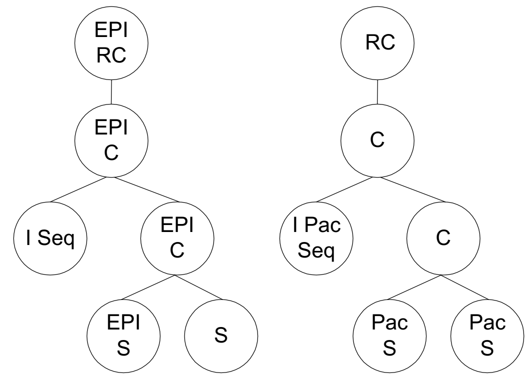

Figure 9. Example combinations of different processors

structure change. Hereby, the possibility to combine epi with non - epi dynamic structure handling processors creates a problem. For epi processing it is sufficient to have epi processors on the path from the epi model to the root (Figure 9, left example). The adaptation of the processor tree can become quite expensive because new processors have to be created and integrated in the existing processing structure and data from the previous processors needs to be transferred. Different strategies to tackle this problem can be imagined: (a) full epi processing - all (coupled-)models are computed by epi processors (b) full adaption - as soon as a path looses the last epi model all its processors are changed (if an epi model is inserted into a path without an epi model the processors need to be changed, too) (c) lazy adaption - once epi the processors will not be converted back, only if an epi model is inserted into a coupled model which before was composed of models without interface to an external process, the coordinator will be adapted and if needed all the coordinators on the path up to the root. The first solution seems to be the most practical, and the additional effort in comparison to a non epi processor seems neglectable.

6. Applications

In [28] the tryptophan synthase is a multi-level model of metabolic processes comprising several thousands of models, most of which are homogeneously structured. Couplings are dynamically added and removed to deliver metabolites to randomly selected enzymes. Currently the model is extended to include the tryptophan operon which regulates the tryptophan production. Model components representing DNA, mRNA, ribosomes, mRNA polymerase, and a set of different enzymes, are involved in a complex interaction: thousands of models are dynamically added and connected to others and models are moving across the model tree hierarchy. These are examples for models that

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 IEEE

comprise several thousands of reactive entities exhibiting changing coupling and compositional structures. No external processes have been invoked and thus the models are executed by the parallel dynPDEVS simulator or its sequential counterpart.

Reactive entities, i.e. mobile agents, are the subject of another application: where we test the possibilities to let mobile agents run in a simulated environment. For that purpose models, so called ambassadors, have been defined interacting with the externally running processes and reflecting crucial changes into the simulated environment [25]. Thus, Mole agents can be executed in the virtual environment as they are executed in their normal run-time environment. Each Mole agent is started by a synchronous invocation of its methods in the simulation. Calls, the sending of messages etc. of the invoked agent's methods are redirected into the simulation environment. Agents frequently migrate from one location to other locations, thus initiating a migration of the ambassadors through the virtual network.

Deliberative entities and with this the invocation of planning systems, are at the core of testing the role of commitment strategies in a dynamic test bed [20]. The planners are invoked synchronously and the time models are used to put more or less time pressure on the planning activities of the agents. Another application analyzes different deliberation strategies to overcome the economic and demographic consequences of disasters in pre-modern towns. The model includes several thousands of utility-based models, e.g. merchants, workers, and craftsmen, and one deliberating actor: the local authority. Here, the time model is constant as only the result of the deliberation process is of interest, rather than how long the local authority of the town does need to come up with a plan [6].

7. Implementational details

By further using the template pattern (see [10, 8]) the effort for creating these new simulators was reduced. The creation of further processors is additionally facilitated by encapsulating the newly added functionality (sc message parsing, passing, and execution as well as the epi handling) in separate classes. Thus, for new processors which shall support sc message or epi handling, we only need to weave the calls to the separate class methods into the existing (inherited) code.

8. Conclusions

By providing a set of different simulator components in JAMES the simulation engine can be easily adapted to the characteristics of the model, the underlying infrastructure, and the users preferences [10]. We developed simulator components dedicated to support variable structure models. One simulator enriches the traditional parallel simulation in DEVS by variable structures and one integrates variable structures into a simulator which supports an interaction between simulation and external processes. Each simulation engine comprises the DEVS typical simulators and coordinators. Whereas sequential and parallel simulators and coordinators can be combined in the simulator tree rather freely, one variable structure simulator requires that all simulators and coordinators have to be able to process variable structures. The new flexibility at the level of the model, i.e. being able to change the structure anywhere in the model hierarchy, implies high adaptation costs at the level of simulators. To reduce the adaptation costs, the simulator tree should be initiated with simulators and coordinators that support as many of the potentially required features as possible. What kind of structural changes can be initiated, is only limited by a model's knowledge about its environment, thus diverse strategies to support variable structure models as implemented in different simulation systems can easily be realized, by providing certain knowledge only to certain models. While the unpaced EPI simulator supports the synchronous invocation of external processes, for an asynchronous interaction between simulation and external processes different strategies are required. An example is the asynchronous communication of JAMES with the AutoMinder software [17]. An already implemented software to remind elderlies shall be tested in a virtual household environment. Therefore we already introduced a first paced epi variant [24]. This variant needs to be refined and to be adapted to the new way of handling structural changes.

9. Acknowledgment

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

References

[1] F. Barros, M. Mendes, and B. Zeigler. Variable devs — variable structure modeling formalism: An adaptive computer architecture application. In *Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems 'Distributed Interactive Simulation Environments'*, pages 185–191, December 1994.

[2] F. J. Barros. Modeling formalisms for dynamic structure systems. *ACM Trans. Model. Comput. Simul.*, 7(4):501–515, 1997.

[3] S. Borland and H. Vangheluwe. Transforming statecharts to devs. In A. Bruzzone and M. Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages 154–159, Montral, Canada, July 2003. Society for Computer Simulation International (SCS).

[4] D. Degenring, M. Röhl, and A. Uhrmacher. Discrete event, multi-level simulation of metabolite channeling. *BioSystems*, to appear, 2004.

[5] S. A. DeLoach. Specifying agent behavior as concurrent tasks. In *Proceedings of the fifth international conference on Autonomous agents*, pages 102–103. ACM Press, 2001.

[6] U. Ewert, M. Röhl, and A. Uhrmacher. *Agent Based Computational Demography*, chapter Consequences of Mortality Crises in Pre-Modern European Towns. Springer Verlag, Heidelberg, 2003.

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

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA, 1994.

[9] C. Gui and P. Mohapatra. Short: self-healing and optimizing routing techniques for mobile ad hoc networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 279–290. ACM Press, 2003.

[10] J. Himmelspach and A. Uhrmacher. A Component-based Simulation Layer for James. In *18th Workshop on Parallel and Distributed Simulation*, Kufstein, 2004. IEEE Computer Society Press.

[11] X. Hu, B. P. Zeigler, and X. Hu. Model continuity to support software development for distributed robotic systems: A team formation example. *Journal of Intelligent and Robotic Systems*, 39(1):71–87, January 2004.

[12] W. Ketter, A. Babanov, and M. Gini. An evolutionary framework for studying behaviors of economic agents. In *Proc. of the Second Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, pages 1030–1031, Melbourne, Australia, July 2003.

[13] M. Köhler and H. Rölke. Modelling mobility and mobile agents using nets within nets. In D. Moldt, editor, *Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents (MOCA'02)*, Technical report of the Department of Computer Science, pages 141–157, Aarhus, Denmark, August 2002. University of Aarhus.

[14] P. Liu and P. Pal, editors. *Proc. First ACM Workshop on Survivable and Self-Regenerative Systems*. ACM Press, 2003.

[15] O. Obst. *Specifying Rational Agents with Statecharts and Utility Functions*, volume 2377 / 2002 of *Lecture Notes in Computer Science*, page 173. Springer-Verlag Heidelberg, 2001.

[16] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.

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

[18] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real - the inhouse card case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2-3):140–157, March 2004.

[19] O. F. Rana and K. Stout. What is scalability in multi-agent systems? In *Proceedings of the fourth international conference on Autonomous agents*, pages 56–63. ACM Press, 2000.

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

[21] T. Takahashi, S. Tadokoro, M. Ohta, and N. Ito. Agent based approach in disaster rescue simulation - from test-bed of multiagent system to practical application. In A. Birk, S. Coradeschi, and S. Tadokoro, editors, *RoboCup 2001*, number 2377 in LNAI, pages 102–111. Springer Verlag Berlin Heidelberg, 2002.

[22] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[23] A. Uhrmacher. Simulation for agent-oriented software engineering. In W. Lunceford and E. Page, editors, *First International Conference on Grand Challenges for Modeling and Simulation*, San Antonio, Texas, 2002. SCS, San Diego.

[24] A. Uhrmacher, M. Röhl, and J. Himmelspach. Unpaced and paced simulation for testing agents. In *Simulation in Industry, 15th European Simulation Symposium*, pages 71–80, Delft, 2003. SCS-European Publishing House.

[25] A. Uhrmacher, M. Röhl, and B. Kullick. The role of reflection in simulating and testing agents: An exploration based on the simulation system james. *Applied Artificial Intelligence*, 9-10:795–811, October-December 2002.

[26] A. M. Uhrmacher. Reasoning about Changing Structure: A Modeling Concept for Ecological Systems. *International Journal on Applied Artificial Intelligence*, 9(2):157–180, 1995.

[27] A. M. Uhrmacher. Dynamic Structures in Modeling and Simulation - A Reflective Approach. *ACM Transactions on Modeling and Simulation*, 11(2):206–232, Apr. 2001.

[28] A. M. Uhrmacher and D. Degenring. From tryptophan synthase to operon: A discrete-event-multi-level approach, November 2003.

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

[30] G. Wainer. `http://www.sce.carleton.ca/faculty/wainer/standard/`, July 2004.

[31] D. Weyns, E. Steegmans, and T. Holvoet. Combining adaptive behavior and role modeling with state charts. In R. Choren, A. Garcia, C. Lucena, M. Griss, D. Kung, N. Minsky, and A. Romanovsky, editors, *Proceedings of the Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, pages 81–90, 2004.

[32] H. Xu and S. M. Shatz. A framework for modeling agent-oriented software. In *The 21st International Conference on Distributed Computing Systems*, page 57. IEEE Computer Society, April 2001.

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

[34] B. P. Zeigler. Devs today: Recent advances in discrete event-based information technology. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, page 148, Orlando, Florida, October 2003.