NESTING SIMULATING AGENTS IN SIMULA

Eugene Kindler
University of Ostrava
Institute for Research and Applications of Fuzzy Modeling
30. dubna street no. 22, CZ – 701 03 Ostrava 1, Czech Republic
E-mail: kindler@ksi.mff.cuni.cz

KEYWORDS
Simulating agents, Object-oriented programming, Nesting models

ABSTRACT

The implementation and application of simulating agents built into simulation models is discussed. They reflect real activities that are rather disseminated namely in designing and operating industrial systems (but outside that domain as well). Some problems and ways for their solutions will be mentioned and certain applications in industry and in health service as well.

INTRODUCTION

Models in Design

When one designs a man-made system (including a natural system enriched by man-made components) he always anticipates its future existence, operation and importance. The designer (a man or a team) uses a certain model – so called ***d-model*** – of such a system, in which he expresses the possible future development (operation, behavior) of the designed system at it. The d-model can be realized in one's mind (possibly with help of paper/pencil etc.) or as running at a computer. The mental one can be more or less exact, based on mathematics, reminiscence, deduction, imagining etc.

The d-models are demanded to be rather true, while the designed systems are rather complex. That is an impulse to convert the mental d-models to those running at computers. The mental d-models based on imagination are often converted to simulation ones, while the other ones use to be far from simulation ones, as they often neglect time or at least its Newtonian flow.

Models in Operation

The man-made systems often suppose humans among their components to manage the system operation. Such humans use also models – so called ***o-models*** – reflecting (in the human mind) the system or its part concerning the one's responsibility. Nowadays, Automation leads to eliminating the human intervention and the o-models are frequently transferred to those implemented to computers. The forceful development of computing technique implies to apply more and more complex o-models. Frequently, the interventions during the system operation concern the system behavior in the more or less near future and the humans often imagine what may happen. Their mental o-models, truly converted into computer ones, result in simulation ones.

Interaction Between Models

During the design, one should take into account all agents that could influence the behavior of the system, after it is let exist and operate. When one supposes an o-model as necessary for the system operation he should include it as a component into the d-model; otherwise the d-model could inform on a system essentially different from the intended one. Or – from the opposite view point – if one believed the d-model to give realistic information on the designed system even in case of neglecting the o-model in it (or replacing the o-model by an essentially simpler one) he would be competent to do it also for the future real system (i.e. to delete the o-model from it or to replace it by the simpler one).

Let us accept a general thesis that a system is composed of elements that mutually interact and that – in case the system is dynamic one – their number can vary in time. And let us also accept that the best computer models reflect the structure (constant or varying) of the systems they model. Programming languages often enable that requirement and namely for the simulation models one can state: the better programming tool is the greater "isomorphism" in the descriptions it admits.

Therefore we are to meet computer models (namely d-ones) among the elements of which there are other models (namely o-models) or carriers of such models (namely humans or computers). The carriers may be simply typified as agents, the dynamics of which is a cycle composed of the following phases:

(a) a passive phase: the agent waits for a signal;
(b) initializing phase, following a signal: the agent detects its environment (actual state of the d-model) and uses the data for building the o-model,
(c) modeling phase: the agent lets the model run,
(d) decision phase: using the data generated by the o-model, the agent derives stimuli and send them to its environment (to other elements of the d-model).

Special Case of Simulation Models

The mentioned agents can be called ***modeling ones***. They carry the o-models and simultaneously are inside d-models; thus we meet a ***nesting*** of models. If an o-model is a simulation one, the modeling agent becomes a ***simulating one***.

First Classification

Naturally, there are four possibilities of such a nesting: the o-model is either a simulation one or not and the d-model is a simulation one or not. But a bit deep view leads us to admit that more modeling agents can be in a d-model; moreover one can suppose that some of them carry simulation models and the other ones carry non-simulation ones. Reflecting the reality, such agents can be present in the d-model at the same time or not and so they could apply the o-models they carry. One should also suppose agents that carry several different o-models; their dynamics consists of *n* occurrences of the sequences of phases like (a)-(d) presented above, while the o-models applied in phases (c) could differ – even so that some of them are simulation ones and the other not.

Let us concentrate our consideration to the case that both d-model and o-models are simulation one. We can speak on nesting simulation. The consideration will be more difficult but the corresponding conclusions can be simply "projected" to the non-simulation cases by neglecting what concerns the simulated time.

AGENTS

Advantage of Agent Attempt

When an o-model is simple, the use of term agent in the expression modeling agent could seem levity. That's true. Nevertheless a simulating agent is never simple and calling it *agent* is serious. But let us study the relations to notion of agent in some detail (see Figure 1).

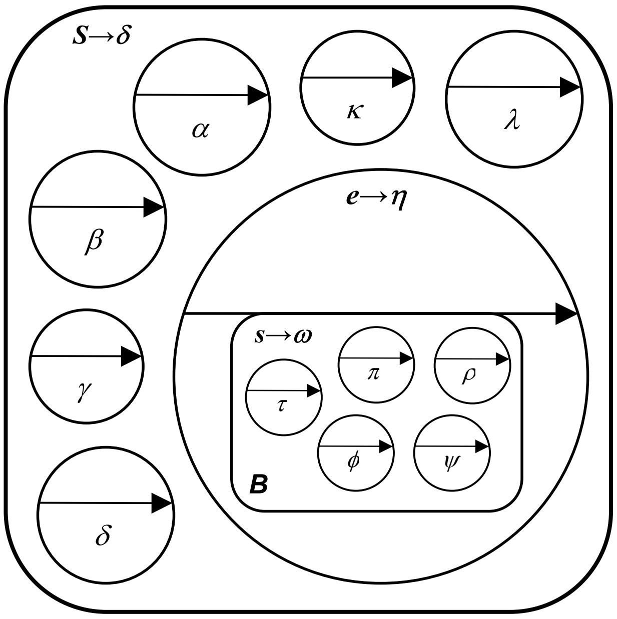

Figure 1: Nesting Agents: The circles represent agents; α – λ are components of δ, π – ψ are components of ω. Arrows symbolize agents' dynamics

A simulating agent η reflects a simulating element *e* of the designed system *S* and in this context it is "isomorphicly" reflected in the d-model δ. In general, *e* interacts with the other elements of *S* and these elements can interact mutually. They are also "isomorphicly" reflected in δ, and the interaction as well. All the elements of *S* (*e* including) are supposed to exist in time in which *S* is supposed to be. In general, to describe "isomorphicly" the dynamics and interactivity of their representations in δ, basic agent-oriented programming tools are helpful. Although degenerated cases (e.g. representations of simple passive elements) are conceivable, the simulating agents underline the real exigency of such tools: η exists in the simulation time of δ and during certain time intervals performs simulation experiments with o-model ω, i.e. it makes complex tasks defended against interruptions from the side of other components of δ.

Thus it is useful to represent all elements of *S* as agents operating in δ. But η itself carries a simulation model ω (o-model) of a system *s* that is also composed of elements; both *S* and *s* are often defined at the same "thing" and the description of ω is similar to that of δ. And thus an idea arises to use the agent tools oriented to the description of δ, for to describe ω, too.

Note that an evident difference between *S* and *s* (and thereupon between δ and ω) may seem: *s* does not need to reflect *e*, i.e. ω should have no simulating agent. From one part, it is often true but the difference does not counter to the advantage to use formally equal agents for representing the same element of *S* and *s*. From the other part, as it will be shown in the conclusion of the paper, sometimes even ω could have a simulating agent.

Agents and Class Instances

Since the beginning of programmable computers, there has been a certain gap between simulation and the other computer domains. It is not generally known that the stimuli to the object-oriented programming (further OOP) have been strictly related to simulation since 1966 (Dahl and Nygaard 1968) and that even in the first design and implementation of the OOP these stimuli were synthesized with other stimuli, also from the part of simulation (so called ***life rules*** of classes, which existed already in the oldest language for discrete event simulation GPSS presented in 1961 and in principles used several decades after – see e.g. (Schriber 1991)) and from general programming development (block nesting introduced in ALGOL 60 – see e.g. (Naur 1961)). While the basic principles of OOP (classes, subclasses, message encapsulation and polymorphism) were accepted by the world programming community (including some simulationists) during the 80ies, the life rules and block nesting return very slowly. Nowadays they are for disposal only in the first OOP tool SIMULA (Dahl et al. 1968), in Beta (Madsen et al. 1993) and a bit in Java.

It is possible to state that the including of the life rules into OOP is a good way for allowing to introduce classes of agents: the life rules ensure their autonomy in operation, the encapsulated methods provide their act-iveness (both outside and from outside) and the poly-morphism facilitates a rather simple formulation of their intelligent behavior; the encapsulated methods can carry stimuli for the dynamics according to the life rules, and the life rules are always governed by a programmable ***switching mechanism*** *Q*, which makes possible formul-ating rich social behavior of the instances.

The block nesting enables the locality of agents. For example, a block *B* can be nested among the life rules of a class *C* and another class *A* can be formulated inside *B*. When the operation of an instance *X* of *C* enters *B*, *X* behaves as an intelligent agent that "thinks", using concept *A* and is able to generate instances of it. A set *N* of several classes like *A* (i.e. those local to *B*) enables *C* to model a system composed of instances of the classes of *N* and thus *C* behaves like an agent able to under-stand a certain theory *T* represented by *N* and to model any system representable in *T*. But furthermore, inside *T*, *X* can apply even any entity (e.g. *Q*) that it can handle outside *B*. Thus *X* may represent an agent that can study its environment *Q*, using *T* as an auxiliary theory.

Suppose *Y* is another instance of *C*. If at that time its dy-namics enters also *B*, *Y* obtains a similar ability as we just described for *X*, but the *Y*'s *T* differs from the *X*'s *T* and so the models that *Y* could base on *T*. It is an image of two intelligent agents that both have some similar in-tellections in their own minds but are mutually indepen-dent and may differ in some opinion. Theoretically, a possibility of serious programming errors exists, con-sisting in transferring an instance *U* of a class local inside *B* of *X*, into block *B* of *Y*. That could be computer model of a curious phenomenon – an imagination of one's mind is put into the mind of another person. Such an error is called ***transplantation*** and one can prove that it can lead to a collapse of the computer carrying the d-model. SIMULA seemed very severe in order to prohi-bit transplantation. Beta behaves similarly. According to some observations, Java could admit transplantation.

When dynamics of an agent leaves *B* the ability of the agent to apply the theory *T* disappears.

A block like *B* can be nested inside a formulation of a method *F* declared for e.g. class *A*. When an instance *X* of *A* enters the block it behaves similarly as described above: namely, it behaves like an entity that "thinks" and uses a theory like *T* when reacting to a message *F*.

NESTING AGENTS

Basic Ideas

A small step is sufficient to accept that a class like *C* is a class of agents, namely the class covering η introduc-ed above. Then any instance of *C* behaves like an intel-ligent agent that is able to think on other agents, which are only represented in its "mind". And another small step is sufficient to transfer the ideas so that class *C* does not concern thinking humans but computers behaving like intelligent agents and possessing certain phases of their operation when they are equipped by an ability to use a formal theory of building and running models formulated upon it. So there is not an essential problem to build simulation models containing simulat-ing agents. A class like *C*, representing a computer, is among the classes of elements forming the d-model δ and when the dynamics of an instance η of *C* enters a block like *B* it represents the simulating phase of η. An occurrence of several instances of *C* would represent a situation that in the d-model an existence of several computers able to carry the models is supposed (why not?) and the transplantation would represent for exam-ple that on instantaneous component (e.g. π) of a certain o-model (e.g. ω) existing in a computer is transferred to be an element of the o-model just existing inside an-other computer (note there are also other hypothetical types of transplantation, namely between an o-model and the d-model).

In the most frequent case of nesting models, both the d-model and o-model concern the same thing and their descriptions are similar. This case is called ***reflective simulation***. In such a case it is advisable to use "the same language" for formulating both the models, i.e. to use the same names in declarations of the corresponding classes figuring in both the models (e.g. inside the simu-lating block like *B*, the description of its local classes uses the same names ocurring in the description of the o-model environment). When the o-model is built in phase (b) it surely has to reflect the state of the d-model and now the names of both the models closely meet. It is the most favorable occasion to make transplantation.

Simula has excellent standard tools for simulation and (especially for scheduling agents, operating in common simulated time axis); therefore they were preferred at simulation centers in Czech Republic. But – in order to be rigorous against transplantation – Simula introduces strict syntactical constraints, and 25 years after its first public presentation it was not known whether it was possible to nest an o-model into a d-model so that both could interact (which is a task essential e.g. in case of reflective simulation). The severity of Simula consists in the form of its switching mechanism: Simula offers it as a standard simulation tool and it is often really good. Its form implies the fact that the description of any simulat-ion model has to be a block and Simula does not admit to get names to blocks. Therefore the o-model cannot figure as an element of the d-model, but only as a cert-ain phase (block) nested in the dynamics of the element that carries it (e.g. η depicted in Figure 1 – the o-model exists only when the dynamics of η is inside block *B*).

The following consequence exists. As it was mentioned for reflective simulation, the use of the same language for describing both d-model δ and o-model ω introduces the fact that the elements of δ and their representations in *ω* have the same names. Although it is agreeable, in phase (b) one should express the statements reflecting that the initial state of *ω* is assigned as the actual state of *δ* (e.g. to tell that *temperature* of the element *G* of *ω* should be assigned by the value of *temperature* of the element *G* of *δ*). But it appears impossible.

Only in 1993 the problem was solved by discovering a certain trick against Simula limitations. But the trick was complicated and its use difficult, and therefore in collaboration between Ostrava University in Czech Republic and Blaise Pascal University in French Clermont-Ferrand sophisticated and complex software was in details studied and prepared to facilitate the reflective simulation (Kindler et al 2001, 2003, 2004).

Model as Agent

A certain break-through came in 2005. The key idea consists in resigning the standard Simula switching mechanism and in formulating a new one. Describing it and its differences from the standard demands special knowledge of Simula statement sequencing and thus let us only mention a basic principle.

Simula offers two different sequencing tools. A so called ***semi-symmetrical sequencing*** is manipulated by a pair *call-detach* statements where *call(X)* expresses that object *X* should take control over the computing until it meets a *detach* statement; then it returns the control to the object that called it. So one object *X* can several time contribute to the computing, performing – may be – its different life rules. A so called ***quasi-parallel sequencing*** in manipulated by the only statement *resume(X)*: it transfers the computing to the life rules of *X* so that *X* has no information from what entity it was so instructed; thus it cannot return the control to such an entity; it can only apply *resume* to another object or to return the control to the block where it is local. A very deep analysis tells that the use of *resume* statements must be very limited and that they cannot be applied e.g. in a form *resume(δ.X)* from the outside of model *ω*, where *δ.X* means "element *X* of model *δ*". The "remote identifying" from outside of a model could lead to transplantation and therefore Simula introduces a limitation that can be expressed in a simple way: the model using quasi-parallel sequencing cannot get its proper identification: it cannot be a class instance but a block (a subblock of its carrier).

The standard switching mechanism of Simula oriented to the corporate behavior based on the common Newtonian simulated time is based on the *resume* statement. To formulate a switching mechanism that would behave in the same way and that would not be based on *resume* statements was a rather hard task; after almost 40 years of Simula existence it was discovered and tested.

The main idea how to implement the new switching mechanism can be simply explained by using metaphors. The original (standard) Simula switching mechanism

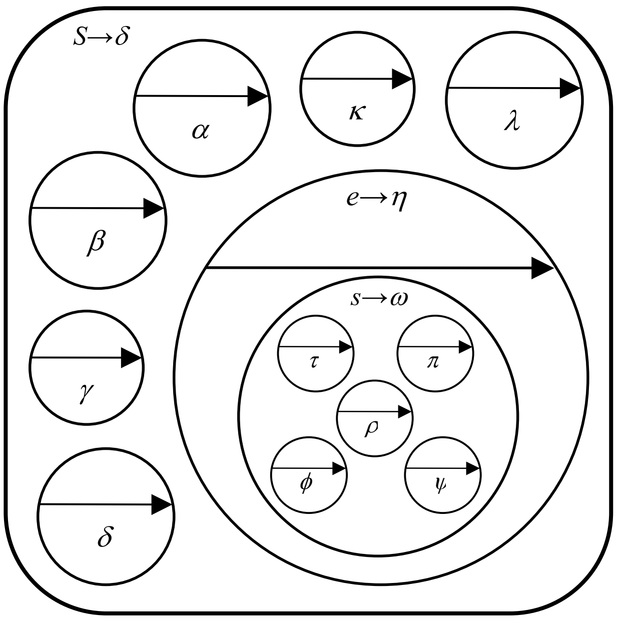

Figure 2: New Conception: o-model *ω* becomes an object and no barrier exists against communication between the objects of *δ* and those of *ω*.

works as a phase of the "simulationist's" life rules; during that phase, this agent has a calendar of events and the agents (processes) composing the simulation model behave as autonomous ones: each of them can communicate with the calendar and possibly send a signal *resume* to another autonomous agent. In the new switching mechanism, only the model itself (not its carrier, i.e. the simulationist) has access to the calendar and according to its actual contents it sends signals *call* to the agents (processes) that compos it; when such a process has to switch the computation to another one, it does not apply *resume* but *detach*, sending a signal to the model, which accepts it and then sends signal *call* to another process.

This technique permits the o-model to be expressed as an object, namely an object of a class local to the object (e.g. computer) regarded as carrying the o-model. See Figure 2. Thus the nested model itself becomes an agent and can get a name. By the way, it is a cheerful surprise that the new switching mechanism is secure against transplantation like the original Simula one!

EXISTING APPLICATIONS

The nesting simulating agents were applied in simulation of maritime harbors controlled by a simulating computer (e.g. Kindler 2000), and in simulation of queuing systems overviewed by one or more simulating dispatchers (Kindler 2001). These studies used the original switching mechanism offered by Simula, which made the programming work very difficult but possible, including the "isomorphic" description of the models (the hard task was to program the "copying" of the actual state of the d-model into the initial state of the o-model).

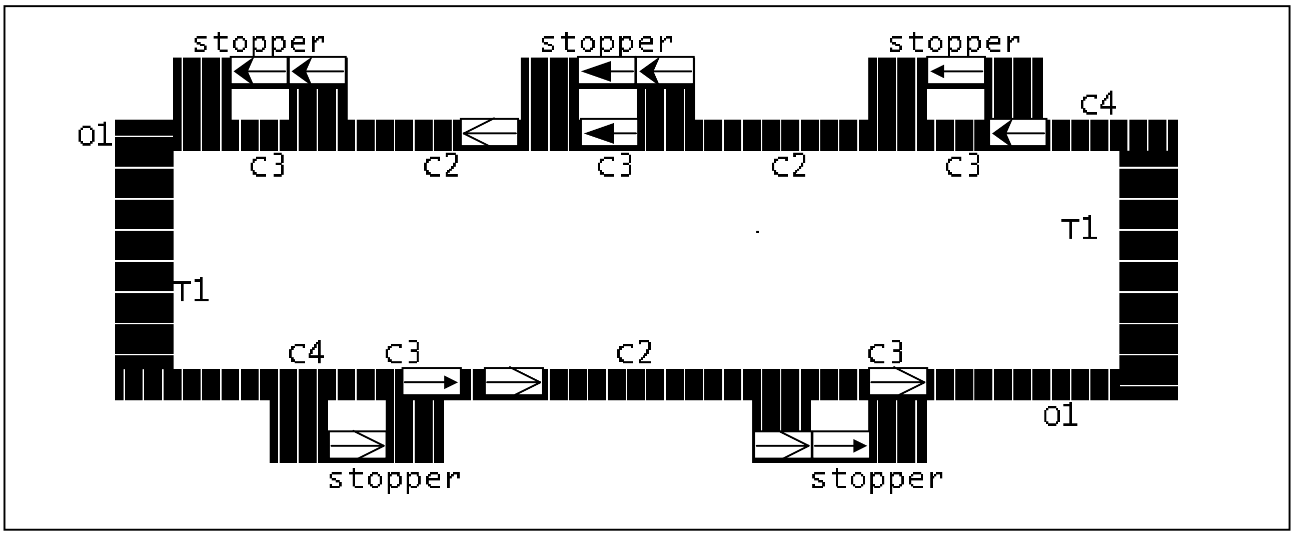

Figure 3: The Conveyor Scheme. The white boxes with arrows represent transported objects, the arrow formats represent the sorts of the membership of the object to a given technological program, and the direction of transporting. The small horizontal sections represent working areas; an object can wait there for processing, be processed (at the center of the working area) and then wait for leaving to the main circle.

The next application concerned local logistics in production systems, namely conveyors with rollers managed by simulating computers (see Figure 3). The first set of models used the original Simula standard switching mechanism (Kindler et al. 2004) and then the models were modified using the new one; their description was simplified, and that enabled producing more models and making them more sophisticated. So the models are able to reflect some decisions concerning the initial decisions connected with the arrivals of the handled objects ("Is it worth to place the object on the conveyor when it is being rather occupied?") and with the unexpected failures ("Is it worth to proceed with a limited number of working areas, and if so, how long?" or "What new logical sequence of technological steps would be optimal for finishing the actually handled series?" (Berruet et al. 2006)).

Other studies were oriented to simulation of service systems, namely (i) for the design of public transport networks in regions of towns and their neighborhoods, applying simulation that should help the passengers by formulating recommendations of the optimal traces, and (ii) for control of demographic development of such regions under assumption that the regions will be facilitated by consulting centers anticipating the future development by using simulation (Bulava 2001).

Into the category of nesting simulating agents, one can include a rather old application of Simula oriented to "continuous" optimizing process during several contemporary simulation experiments (Weinberger 1987). The basic simulated system is viewed as composed of agents that should reach the parameters of the optimum of a certain system; each of them starts with a certain own hypothesis on the parameters and with them he simulates the system. During the simulation the agents iterate a certain "discussion" so that any of them can accept recommendations and findings of his colleagues and accordingly modify his own parameters. During the session, the parameters are drawing near their optimum configuration so that at the end of the session they are good approximation of it. The simulated "discussing agents" are also simulating and use simulating agents inside the models they use for testing their own variants.

At the present days, there were performed opening steps in implementation of models of in-patients flow in hospitals that frequently use simulation to anticipate the possible consequences of their instantaneous decisions (Krivy and Kindler 2006). The models use the new switching mechanism.

CONCLUSION AND FURTHER CLASSIFICATION

The nesting agents enable implementing d-models so that the o-models nested in them contain also an image of a simulating element. Therefore several levels of nesting are possible. The first steps were performed without the new switching mechanism. They modeled idealized systems of competing enterprises *G* and *H* that use simulation to anticipate the rival's future reactions, but the simulationist *f(G)* working for *G* was represented to simulate his "colleague" *f(H)* so that *f(G)* included the o-models supposed at *f(H)* into his own o-models (Blümel and Kindler 1997). The new switching mechanism promises to simulate many similar systems.

That is a stimulus to add further classification criteria to those mentioned at the beginning of the present paper. The criteria concern the "depth" of nesting, i.e. the number of nesting levels. Another criterion offers, namely that according to the presence of reflective simulation in the "tree" of nesting models.

The models using the new switching mechanism work a bit slower than those using the mechanism offered as standard one. Among the next work there is the analysis of the Simula compilation process with a target to make the models using the new switching mechanism as fast as possible.

ACKNOWLEDGEMENT

This work was supported by the Ostrava university research scheme MSM6198898701.

REFERENCES

Berruet P. and E. Kindler. 2006. "Nested Anticipation in Design of Reconfigurable Manufacturing Systems". *IJCAS (International Journal of Computing Anticipatory Systems)*, in print

Blümel, P. and E. Kindler. 1997. "Simulation of Antagonist Mutually Simulating Systems". In: *Simulation und Animation '97,* O. Deussen, P. Lorenz (Eds.). Society for Computer Simulation International, Erlangen, Ghent, Budapest, San Diego, 56-65

Bulava, P. 2002. "Transport System in Havirov". In *Proceedings of 28th ASU Conference* (Brno, Sept. 26-30). Technical University, Brno, Czech Republic, 57-62.

Dahl, O.-J. and K. Nygaard, 1978. "Class and Subclass Declarations" . In *Simulation Programming Languages*, J. N. Buxton (Ed.). North-Holland, Amsterdam, 159-174

Kindler, E. 2000. "Nesting Simulation of a Container Terminal Operating With its own Simulation Model". *JORBEL (Belgian Journal of Operations Research, Statistics and Computer Sciences)*, 40, No.3-4, 169-181

Kindler, E. 2001 "Computer Models of Systems Containing Simulating Elements". In: *Computing Anticipatory Systems CASYS 2000 – Fourth International Conference,* (Liège, Belgium, Aug., 7-12 2000). AIP (American Institute of Physics), Melville, New York

Kindler, E., T. Coudert and P. Berruet. 2004. "Component-Based Simulation for a Reconfiguration Study of Transitic Systems, *SIMULATION*, 80, No. 3 (March), 153-163

Kindler, E., I. Krivy and A. Tanguy. 2001. "Reflective simulation of discrete logistic and production systems". In *Modeling and Simulation 2001, 15th European Simulation Multiconference ESM2001* (Prague, June 6-9). Society for Computer Simulation International, Delft, 861-865

Kindler, E., I. Krivy and A. Tanguy. 2003. "Simulation of Simulating Computerized Systems". In *International Workshop of Modelling & Applied Simulation MAS 2003* (Bergeggi, Italy, Oct. 2-4). Liophant Simulation Club, DIP – Genoa University, McLeod Institute of Simulation Science – Genoa Center, Genoa, 16-21

Kindler, E., I. Křivý and A. Tanguy. 2004. "Object-Oriented Sytem Analysis of Anticipatory Systems in Week Sense". *International Journal of Computing Anticipatory Systems*, 14, 271-285

Krivy, I. and E. Kindler. 2006. "Synthsis of two Anticipatory Modes in Design and Life-Sycle of Hospitals". *IJCAS (International Journal of Computing Anticipatory Systems)*, in print

Madsen, O. L.; B. Møller-Pedersen; and K. Nygaard. 1993. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Harlow – Reading – Menlo Park

Naur, P. (ed.), 1963. "Revised Report on the Algorithmic Language ALGOL 60." *Communications of the ACM*, 6, No.1 (Jan), 1-17.

Schriber, T. S. 1991. *An introduction to Simulation Using GPSSH.* Wiley, New York

Weinberger, J. 1987. "Extremization of Vector Criteria of Simulation Models by Means of Quasi-Parallel Handling", *Computers and Artificial Intelligence*, **3**, 71-79

EUGENE KINDLER was born in 1935 in Prague (Czechoslovak Republic). There he studied mathematics at Charles University and got Doctor of philosophy in Logic and Doctor of sciences in Mathematics grades. Czechoslovak academy of sciences appointed him the grade of Candidate of sciences in physics/ mathematics. During 1958-1966 he worked eight years at the Research Institute of Mathematical Machines in Prague, participating in designing the first Czechoslovak electronic computer. He is also the author of the first Czechoslovak ALGOL compiler. Then he worked seven years with the Faculty of General Medicine of Charles University where he developed a computing center oriented to nuclear medicine and radiobiology. He developed the first Czechoslovak simulation language called COSMO (Compartmental system modeling) and its compiler. Beginning from 1974, he worked with the Faculty of Mathematics and Physics of Charles University. Since 1993 he has collaborated with a newly established University in Ostrava (Czech Republic) in order to develop there a team for modern object-oriented simulation. After his retirement at Charles University, he came to the Ostrava University, working there as Professor of applied mathematics. Having appreciated the first real stimuli coming in 1967 from simulation to the programming style that was later called object-oriented programming, Kindler founded the object-oriented methods use in Czechoslovakia in the late sixties. Nowadays his main interest concerns object-oriented simulation of systems controlled by simulating computers they contain. He participated at projects concerning steel production scheduling and transport optimizing in agriculture and in machine production. He worked as visiting professor with University of Pisa (Italy) and with West Virginia University in Morgantown (USA) and as invited professor with Blaise Pascal University in Clermont-Ferrand and with University of South Brittany in Lorient (France). During the last decade he participated at projects supported by European Commission and oriented to modernizing maritime harbors.