SIMULATION OF INTELLIGENT SYSTEMS – BLOCK STRUCTURE REVISITED

Eugene Kindler
Ivan Krivy
University of Ostrava
Department of Informatics and Computers
CZ – 701 03 Ostrava, Dvorakova 7
Czech Republic
E-mail: ekindler@centrum.cz
ivan.krivy@osu.cz

KEYWORDS
Agents, Local Agents, Nested Agents, Simulation, Object-oriented programming, SIMULA

ABSTRACT

In this paper essential aspects of block-oriented world viewing are emphasized and presented in a relation to simulation models of systems managed by complex and sophisticated decision rules. The construction of such models can be essentially simplified when one combines the agent-oriented, object-oriented and block-oriented paradigms of programming. The use of the described way leads to certain paradoxical relations among computing agents. Some applications are added.

BASIC SUPPOSITIONS

Main Technique of Constructing Simulation Models

The technique of computer simulation is commonly applied in case the studied system is complex. The consequence is that the simulation models are complex, too, and there are difficulties with their programming and debugging. Since the early sixties, the simulation programming languages have enabled evading such difficulties. Instead of algorithmizing the run of the simulation models, the users of these languages had only to describe the simulated system and the language processor automatically translated such descriptions into corresponding algorithms (computer programs).

Already one of the first simulation languages, GPSS, offered their users to declare certain sets of elements equipped by similar configurations of their private entities (called "parameters") and influencing the computer run by commonly formulated "life rules", i.e. algorithms switched according to a common "scheduling system" related to common modeled Newtonian time flow (Gordon 1961, Florea and Kalisz 2000).

The first object-oriented language SIMULA 67 (Dahl and Nygaard 1968) introduced also the life rules, generalized them to "co-routines" and led to the agents. In spite of they essential help it to be successful in simulation of very large systems, the next popular existence of the object-oriented programming (in C++ and Smalltalk) refused the life rules. Long time after, the agent orientation slowly returned, but under other titles and other metaphors taken from the world around us.

Principles of Block Orientation

True block orientation was introduced in ALGOL 60 (Backus et al. 1960), considering blocks as parts of algorithms with own entities (namely "local" variables, procedures and – for some good authors also destinations of transfers in algorithm run). In fact, such parts of algorithms, called "textual blocks", are only abstract patterns of "block instances", i.e. components of the relating computing process. The block instances are in a certain case "copies" of the textual blocks, which are able to influence the computing according to the statements occurring in the related textual blocks. In some cases (e.g. recursive calls of procedures) more instances of the same textual block exist in the same time and influence the computing process.

An essential aspect of the block orientation is block nesting: Textual block *b* is considered as a certain sort of statement and occurs among the other statements forming another textual block *B* (and so on recursively, as *B* can be also among the statements of another block). In such a case, *b* is called ***nested*** in *B*. The consequence of this nesting is that the statements (operations) of the nested block *b* can manipulate not only the entities local in it but also in the block *B* in that it is nested (and so on recursively).

Block nesting carries a problem of "name conflict". If an entity accessible under name e.g. *N*, is local in a block *b* nested in block *B*, and if an entity accessible under the same name *N* exists also in block *B*, then a question arises how to interpret name *N* incident in the statements of block *b*. The common rule, respected both in ALGOL 60 and in SIMULA 67 tells that in such a case the name *N* is interpreted as that of the entity introduced in the "nearest" block in the hierarchy of nesting, i.e. as that of the entity local to *b*. In such a case, the entity introduced under name *N* for block *B* is not accessible by the statements occurring in block *b*.

Proceedings 22nd European Conference on Modelling and Simulation ©ECMS Loucas S. Louca, Yiorgos Chrysanthou, Zuzana Oplatková, Khalid Al-Begain (Editors)
ISBN: 978-0-9553018-5-8 / ISBN: 978-0-9553018-6-5 (CD)

Algol 60 and SIMULA 67 do not allow to give names to instances of blocks. Nevertheless, according to the definition of SIMULA 67 the class declarations are viewed as analogies to blocks or – more exactly – as procedures and the instances of classes are interpreted as block instances, differing from the other block instances so that they can get names during the computing run.

Synthesis of Block Orientation with Object Orientation

The declaration of a class has many similar aspects as the declaration of a variable or a procedure and thus SIMULA 67 introduced the rule that a class can be declared as a component of another class or as a local entity of a block. The first variant can be considered as "direct" nesting of a class in another class. Let class *c* be local in a block that occurs among the statements forming life rules of another class *C*. That is an example of another class nesting, which we could call "indirect" one.

The consequence of the fact that class instances can get names is, that an instance *b* of the mentioned class *c* can get a name e.g. *R1*, and an instance *B* of class *C* can get a name e.g. *R2* different from *R1*. If a local entity in *C* is introduced under name *N*, and another local entity in *c* is introduced under the same name *N*, both the entities can be distinguished by using expressions telling "*N* of *b*" and "*N* of *B*".

Note that the concept of block is related to that of statement. In other words, a synthesis of block orientation and object orientation is fully meaningful, only when also agent orientation enters the same synthesis. Nowadays, one could mention BETA (Madsen et al. 1993) and – with a certain shutting eyes – JAVA, as certain examples of programming languages based at the synthesis of both three mentioned orientation.

SYSTEMS WITH COMPLEX AND SOPHISTICATED DECISION RULES

Internal Models

Many systems are viewed as complex not only for they are composed of many elements of many sorts differing by the rules of their mutual interactions, but also that the interactions are themselves complex even in case such an interaction may exist among few elements. Examples are systems in that one or more humans operate and make decisions that influence not only them but also their environment, which can cover the whole system.

At the present time, the human decisions are replaced (and/or modeled) by computer activities. In such a case, the capacity of computation and storing information enables the computer to analyze the instantaneous state of the given system and to optimize the system's reaction to it much better than a human being could do that. This situation leads to the task of constructing computer models of the human decisions. In principle, the models do not need to be simulation ones. Nevertheless, human often makes a decision oriented to a certain profit in future. In other words, a system with such a human "factor" is an anticipatory system according to a definition formulated by (Rosen 1985) or anticipatory system in a week sense according to a later definition by (Dubois 2000). Humans often use imagining what could happen as a sequence (or tree) of consequences of such a decision. According to the imagined events, the concerned human evaluates his/her possible decisions and chooses the optimum one.

The computer analogy of such an imagining process is a true simulation model. Therefore, if we simulate a system *S*, the dynamics of which is influenced by decisions made by element *C* that uses a model *m*, then *m* should be an instance of a class *s* nested in an instance *E* of class *c* reflecting the element *C*; and *c* should be nested in an instance *M* of a class Σ of the models of *S*. Using diagrams designed e.g. in (Mejtsky and Kindler 1980, 1981), this network of *nesting* relations is illustrated in Fig. 1, so that the relation "*x* is nested in *y*" is represented by the relation "circle-image of *x* is inside the circle-image of *y*". Note that the described nesting fully represents the access relations existing in system *S*, where *m* has access to the contents of the element *C* inside that it exists, and – having use of *C* – *m* has access to every element of system *S*. (The names of the circles that represent the instances are not in brackets, the corresponding classes are in the square brackets and the names of the corresponding modeled "things" are in the round brackets.) The circles in the dashed lines, i.e. *H1*, …*H7*, represent other elements of the model *M*, i.e.

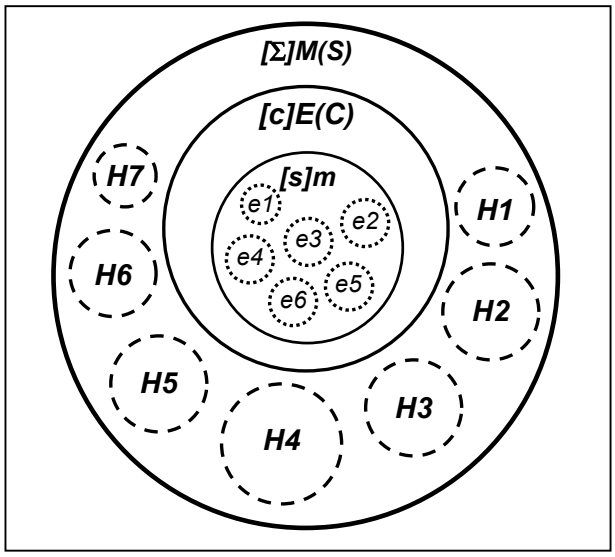

Figure 1: Mejtsky's diagram of the described case

models of components that exist in *S* currently with *C*; the small circles in dotted lines, i.e. *e1*,…, *e6,* represent the elements of model *m*, i.e. no "hard physical components" of *S*.

Examples

Not later than in 1990 (Kindler and Brejcha 1990), a production system containing automatically guided vehicles was simulated as follows: when a vehicle asked for its trace, it computed the shortest way by simulating a fictitious system of propagated pulse, according to an idea developed by Lee and by Dijkstra (see Dahl 1966). A similar technique was applied later, namely in simulating container yards with internal operation transport performed by ground moving vehicles (Kindler 1997). In the first case, the vehicles were viewed as really automated, while the second case started with a conception to model the intelligent decisions of human drivers, but the task of the studies was to transform human thinking to computer automating with a prospectus to have use of the computer that would control the whole system of the yard – the capacity of its memory would enable considering also the information hidden for the drivers' sights and minds. In both the cases, the simulation models reflected systems containing elements that embody a certain intelligent behaving in a labyrinth of possible paths. The intelligent decisions of the elements was based on models "privately used by the elements" and more or less reflecting the instantaneous state of their environment and the demands for future.

The last case was later completed by including anticipation of possible conflicts with other elements. The vehicles had other "privately used" models that simulated the following use of computed paths and so discovered the possible barriers that a vehicle could prepare to another one (Kindler, 2000). (Novak 2000) performed a further step in the development by synthesizing both the models in order to introduce more complex movements of the vehicles (with returns and mutual keeping away, in order to allow more complex computing shorter paths). In that way, a model of a vehicle obtained an ability to view, anticipate and improve the future not only of itself but of other vehicles as well.

The same technique was used by (Bulava 2002) for the simulation of public transport by busses in a certain North-Moravian region. The model reflected the intelligent behavior of the passengers; in general, each of them envisions several possible combinations of the buss lines to access his target, and accordingly chooses the best one. In (Bulava 2007), the same author applied a similar way for simulation of demographic development of a region. His model contains analogies of the processes of the decisions of the humans about changing their states (like residences, marital states, children, schools, employers), based on their envisioning and/or using simulation models installed in consulting rooms.

In cooperation with French specialists in designing configurations of industrial conveyors, tailored according to the particular user demands, the simulation of "intelligent" cyclic-form conveyors was started (Kindler et al. 2004). Several simulation models were nested into the whole model of the conveyor; they served as follows:

(a) when a new parcel *P* comes to be served by the conveyor: to anticipate the dynamics of the parcels present at the conveyor and accordingly to decide whether *P* should be placed onto the conveyor immediately or later;

(b) when a parcel *P* was just processed at a certain working area and a choice among several working areas for the future processing of *P* exists: to anticipate the state of each of the plausible working areas at time when it would be accessed by *P* and accordingly to decide on the best target for it;

(c) in case a fault comes so that a certain working area is not accessible by the conveyor: to anticipate whether the current orders (sets of parcels) can be processed by the remaining working areas and accordingly to decide whether to continue or to stop immediately the conveyor and repair it;

(d) when the decision in case (c) is to stop immediately the conveyor: to try continuing so that the technological programs for various orders are modified, i.e. to anticipate what could happen when each of them were applied, and possibly to determine the optimum choice of new programs.

A bed ward in a hospital is a dynamic system influenced in certain states by some employees who could decide e.g. on putting together patients placed in several rooms or on the room to place a patient there (in case there are more rooms where the patient could be placed). The intelligence of such decisions could concern anticipating their possible consequences. That was included into simulation models of bed wards (Krivy and Kindler 2006) but until nowadays without particular applications.

In the last months, one makes the first steps in simulation models of reconfigurable computing networks that contain one or more computers, the function of which is to modify the links among the other computers with due regard partly for the instantaneous state of the network and partly for its possible future development (Kindler et al. 2007).

DEPENDING AND SIMULATING AGENTS AND AGENTS-MODELS

At conference Agent-Based Simulation (Urban 2000), an interesting discussion took place whether the conference title means simulation with use of agents or

simulation of agents. Since that time, the further development has shown the "agent-basing" being more complex.

Already GPSS, which represented the first paradigm of "embryonal stage" agents shown the possibility of simulation by means of agents. The computing agents were not too autonomous but they were almost "isomorphic images" of the "material agents" existing in the simulated system and in that manner they could be formalized by the GPSS users. The lack of autonomy of the agents served to support their social behaving, namely in a similar way as that of their material origins.

Let such agents be called ***depending*** ones. While in GPSS their social behaving was focused around common system time flow and queue discipline, the further development lead to offer the user tools for expressing other aspects and events in social behaving, like mutual exchange of data (including the case that an agent's behavior could be switched in the dependence on any component of the instantaneous states of other agents). It was developed into the most perfect way in the inspection constructs (connection statements) of the "old SIMULA" (SIMULA I), i.e. in the discrete event simulation language implemented in 1964, to that time still without object-orientation (Dahl and Nygaard 1966). The further language SIMULA (formerly called SIMULA 67) introduced – beside the object orientation – also "dot notation" (remote identifying) that allowed complete freedom in expressing interactions among the depending agents (Dahl and Nygaard, 1968).

As it was mentioned above, the block structure of the object-oriented SIMULA (i.e. SIMULA 67) allowed to formulate another type of agents, which could be called ***simulating*** ones. When the dynamics of a depending agent *D* enters a block with local simulated time and local class of depending agents, the block represents a simulation experiment that is to be performed while *D* operates inside the block. Thus a depending agent (to a certain model *M*) can become a simulating one, applying a model *m* that differs from *M* and exists only as some "information process" linked to *D*. When the dynamics of *D* entered the mentioned block and *D* became simulating agent, it does not cease being depending one, and when its dynamics leaves the block, *D* ceases being the simulating agent and further exists only as depending one. Naturally, the dynamics of *D* can perform e.g. a cycle and return to the block and so *D* could repeat being simulating agent. Moreover, the dynamics of *D* can contain more than one block of the given properties, and so *D* could be considered to handle different simulation models. An example is offered in the simulation of container yard presented above, where any vehicle is represented by a depending agent, which handles one model when computing the shortest path, and another model when testing it against barriers. In Fig. 1, *H1*, …, *H7* and *E* can be viewed as depending agents forming *M*, while *E* is depicted there as being just simulating agent and handling model *m* composed of its own depending agents *e1*, …, *e6*.

The described structure can be generalized so that in the dynamics of *D* the mentioned block need not be equipped by simulated time flow and the classes local in the block need not be related to such a time flow. Then the block represents a model that is not simulation one and *D* could be called ***modeling agent***.

The standard tool for simulation offered by (the object-oriented) SIMULA as its standard class called *SIMULATION* does not allow to identify simulation model. The reason is to provide the program product against illogical mixing of models; it is difficult to locate such a programming error when it occurs. The consequence of the restriction is that no simulation model can be named and thus handled as an object and assigned from one depending agent to another one. We discovered and implemented a class called *SIMULAT* that is not restricted in the mentioned manner (Krivy and Kindler 2007). Using this class, the model itself becomes an agent, which one can call ***agent-model***. In the real world, a simple interpretation of the changing of assignments of an agent-model is changing models among different computers. In Fig. 1, the agent-model *m* is depicted as assigned to depending agent *E*. Note that a depending agent that is just simulating is behaving similarly as if an agent model would be linked to it.

CONCLUDING REMARKS

Application Inducement

As it was mentioned in section Basic Suppositions, the most suitable situation to construct a simulation model of a complex system consists in describing the simulated system and having an occasion of automatic conversion of the description into the program the run of that would realize the computer simulation experiment and – possibly – a simulation study in case the program is e.g. repeated as a subprogram in a cycle. Let us return to that idea and analyze it in details. As illustration of the analyzis let us use examples oriented to transport.

The usual process of abstraction from reality to systems about that one decides to simulate them concerns interactions of their elements, which do not directly depend of their possible individual abilities to process information in complex manner; such interactions can be viewed as "material" or "physical". In case of transport systems, their examples are moves (changing position, carrying, loading, unloading, manipulating other elements), duration of such interactions, impenetrability of mass objects, local destination, etc. They can be simply built into the life rules of transport tools and sometimes such life rules are not much more than a formalization of such interactions (namely when they

operate by use of some data sets that complete them – like the paths and the destinations). Such systems reflect parts of the reality that react to the given data and to the instantaneous environment rather bluntly, but they are often in focus of simulation, as a large system of such bluntly interacting elements (irrelevantly whether transport tools, customers in a department store, pests or molecules) may offer hard difficulties to techniques different from simulation.

To express such life rules is a natural reflection of what we view at the dynamics of the elements and what we classify according to the sort of them. Thus it is rather simple to express the life rules for different classes as algorithms composed of cycles, assignments, branchings and sending messages to other elements, and sometimes interleaved with scheduling statements. The life rules make the instances depending agents; the classes themselves are to occur out of them and may be considered as figuring at the level of the block containing the description of the whole system (i.e. representing the whole simulation system).

Now let us assume the task is to model a certain class *H* as a class of "intelligent" elements, e.g. that we have to reflect the fact that the driver of any transport tools is able to determine its path and to perform a bit a criticism of what he determined. Determining the path, he thinks and uses some concepts that do not need to reflect something of his environment but that are partially related to this environment. In system abstraction, every driver can be unified with the transport tool he drives and the rules for the life of the transport tool can be understood as those for the driver. A natural way to model the driver's thinking is to insert a block *B* at the given place of the mentioned life rules, so that the private concepts used by the driver are set as local in this block. *B* can be viewed as a good image of the driver's "state of mind", in which some concepts occur. Note that if the life rules of two drivers are in *B* simultaneously, each "life" generates its own instance of *B*, which is a good image of the fact that if two persons think in a similar way they use their own means with their "privat" interpretations of the concerned concepts. As *B* is nested in the life rules of a driver, all entities introduced for the driver's life rules are accessible in *B* and, therefore, the description of the computing inside *B* can be described also in natural way – the driver perceived his environment, determines the path and interprets it as data assigned for something that is not limited to *B*.

Let us abandon the driver's psychology and assume a computer *C* replaces it – irrelevantly whether it is in the transport tool or in some remote but accessible place. Then block *B* can represent the corresponding process of *C* (e.g. applying Lee/Dijkstra method, as mentioned above in section Examples). The technical aspects of links between the transport tool and *C* may be simply mapped by using a metaphor as if *C* was a part of the driver's mind (in system abstraction, only the corresponding task of *C* can be so interepreted), which enables to understand the metaphor that two instances of *B* exist inside the minds of two different drivers.

The thinking of a driver can continue by a criticism whether the computed path is suitable. This represents a new phase of the driver's thinking and, therefore, it can be naturally represented as another block *B** nested in the same life rules at a place following *B*. Other concepts figure in that phase of the driver's thinking and. therefore, other classes may occur in *B**. If a computer *C* is used to make the criticism in a better way than the human mind, i.e. if is used to simulate the just computed path in order to test whether it will not lead to a conflict (see section Examples), the whole simulation model used for that purpose can be described in *B** together with the used classes. The classes and other entities introduced in *B** can be named in the same manner like those used for the description of the whole model; the reason consists in the rule for solving the name conficts (see section Principles of Block Orientation). Namely, when a name *N* is introduced for the whole model and in *B**, too, then *N* occurring inside *B** is interpreted as a name of an entity local to *B**, while *N* occurring outside *B** is interpreted as a name of an entity that coud occur in the whole model. Note that "dot notation" of SIMULA offers access to such an entity also from inside of *B** and, therefore, one can simply describe what makes a driver (possibly "amplified" by *C*), when he is constructing a copy of entity *N* of his environment so that the copy exists in his own mind and is also called *N*.

PROSPECTUS

The authors' further plans concern preparing SIMULA classes that would enable programming depending agents as completely autonomous program modules so that their environment and social behavior would be defined by means of a single pointer to "world". The tools for module compilation, for virtuality (not only of the methods but also of the targets of transfers in the dynamics) and for the branching in the inspections (Simula Standard 1989) make that idea real.

ACKNOWLEDGEMENTS

This work has been supported by the Grant Agency of Czech Republic, grant reference no. 201/060612, name "Computer Simulation of Anticipatory Systems".

REFERENCES

Backus, J. W. et al. 1960. “Report on the Algorithmic Language ALGOL 60.” Numerische Mathematik 2, 106-136.

Bulava, P. 2002. “Transport system in Havirov”. In *Proceedings of 28th ASU Conference* (Brno, Sept. 26-30). University of Technology, Brno, 57-62.

Bulava, P. 2007. *Nested Simulation of Demographic Processes* (in Czech). Doctoral thesis, University of Ostrava, Ostrava.

Buxton, J. N. (Ed.). 1968. *Simulation Programming Languages Proceedings of the IFIP working conference on simulation programming languages, Oslo, May 1967*. North-Holland, Amsterdam 1968.

Dahl, O.-J. 1966. *Discrete Event Simulation Languages*. Norwegian Computing Center, Oslo. Reprinted in (Genuys 1968), 349-395.

Dahl, O.-J.; and K. Nygaard. 1966. “SIMULA – an ALGOL-based Simulation Language”. *CACM* 9, No. 9 (Sept.), 671 -678

Dahl, O.-J.; and K. Nygaard. 1968. “Class and Subclass Declarations.” In (Buxton, 1968), 158-174.

Dubois, D. 2000. “Review of Incursive, Hyperincursive and Anticipatory Systems – Foundation of Anticipation in Electromagnetism.” In *CASYS'99 – Third International Conference,* D. Dubois (Ed.). The American Institute of Physics, Melville, New York, 3-30.

Genuys, F. (Ed.). 1968. *Programming Languages*. Academic Press, London – New York.

Gordon, G. 1961. “A General Purpose Simulation Program.” In *Proc. 1961 EJCC*. MacMillan, New York, 81-98.

Kalisz, E. and A. M. Florea. 2000. “A GPSS simulation model of interactions in a market-based multi-agent system.” In: (Urban, 2000), 145-150.

Kindler, E. 1997. “Classes for object-oriented Simulation of Container Terminals”. In *Managing and Controlling Growing Harbour Terminals*, E.Blümel (Ed.). The Society for Computer Simulation International, San Diego, Erlangen, Ghent, Budapest, 175-278.

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. and M. Brejcha. 1990. “An application of main class nesting – Lee's algorithm.” *SIMULA Newsletter*, 13, No. 3 (Nov.), 24-26

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.; C. Klimes; and I. Krivy. 2007. “Simulation Study With Deep Block Structuring”. In *Modelling and Simulation of Systems MOSIS 07,* J. Stefan (ed.). MARQ, Ostrava, 26-33.

Krivy, I. and E. Kindler. 2006. “Reflective Simulation of In-Patients Dynamics.” In *5th International Conference Aplimat* (Bratislava Febr. 7-12). Part 1. Slovak University of Technology, Bratislava, 613-617.

Krivy, I. and E. Kindler. 2007. “New Flexible Simulation Tool in SIMULA.” In *ESM’2007 – The 2007 European Simulation and Modelling Conference Modelling and Simulation ’2007* (St. Julian’s, Malta, Oct. 22-24). EUROSIS-ETI, Ghent, Belgium, 124-128.

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

Mejtsky, J.; and E. Kindler. 1980. “Diagrams for quasi-parallel sequencing – Part I”. *SIMULA Newsletter* 8, no. 3 (Nov.), 46-49

Mejtsky, J.; and E. Kindler, 1981. “Diagrams for quasi-parallel sequencing – Part II”. *SIMULA Newsletter* 9, no. 1 (Feb.), 17 -19

Novak, P. 2000. **“**Reflective Simulation with Simula and Java”. In *Simulation und Visualisation 2000*, T. Schulze, V. Hinz, P. Lorenz (Eds.). The Society for Computer Simulation International European Publishing House, Ghent, 183-196.

Rosen, R. 1985: *Anticipatory Systems*. Pergamon Press, New York.

Simula Standard. 1989. SIMULA a.s., Oslo.

Urban, Ch. (Ed.). 2000. *Agent-Based Simulation. Proceeding of Workshop 2000, Passau, Germany, May 2000*. The Society for Computer Simulation International, San Diego.

AUTHOR BIOGRAPHIES

EUGENE KINDLER was born in 1935 in Prague (Czechoslovak Republic). He studied mathematics at Charles University in Prague and there he got grades of Doctor of philosophy in Logic and Doctor of sciences in theory of programming. The Czechoslovak academy of sciences granted him the grade of Candidate of sciences in physics/mathematics. During his employment in the Prague Research Institute of Mathematical Machines (1958-1966), he participated at the design of the first Czechoslovak electronic computer and designed and implemented the first Czechoslovak ALGOL compiler for it. Then, working at the Ĭnstitute of Biophysics at the Faculty of General Medicine of Charles University (1967-1973), he designed and implemented the first Czechoslovak simulation language and then introduced the object-oriented programming into Czechoslovakia. Nowadays, as professor emeritus of applied mathematics, he teams up with University in Ostrava. He was visiting professor at the University in Italian Pisa, at the University of South Brittany in French Lorient, at Blaise Pascal University in French Clermont-Ferrand, and at West Virginia University in American Morgantown. His main interest consists in object-oriented simulation of systems that handle complex models to improve their own anticipatory abilities and intelligence.

IVAN KRIVY graduated at the Czech Technical University (1962) and the Charles University (1974). In 1975 he obtained the degree Ph.D. in solid state physics. He worked 13 years at the Nuclear Research Institute (1962-1975) and later (1975-1977) at the Computer Art Establishment as a system programmer. Since 1977 he was teaching courses in programming, probability theory and statistics at the Pedagogical Faculty at Ostrava. After the University of Ostrava having been established in 1991, he entered the Mathematics Department of the Faculty of Science and nowadays he works at the Computer Science Department as Professor in applied mathematics. Since 1992 his research activities are oriented to dynamic modelling and computer simulation as well as to stochastic algorithms and their use in the global optimization. Krivy is a member of the American Mathematical Society, the Association of SIMULA Users, and the International Association for Statistical Computing. He is author of about 190 publications including 4 monographs. Stays abroad: Silesian University of Katowice, Portsmouth Polytechnic, University of Wroclaw, Universidad de Cordoba, Lulea University of Technology and Blaise Pascal University of Clermont-Ferrand.