International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:9, 2007

Simulation and 40 Years of Object-Oriented Programming

Eugene Kindler

Abstract**—**2007 is a jubilee year: in 1967, programming language SIMULA 67 was presented, which contained all aspects of what was later called object-oriented programming. The present paper contains a description of the development unto the object-oriented programming, the role of simulation in this development and other tools that appeared in SIMULA 67 and that are nowadays called super-object-oriented programming.

Keywords**—**Simulation, super-object-oriented programming, object-oriented programming, SIMULA.

I. Introduction

THE present year is a jubilee year. 40 years ago, the detailed properties of the programming language SIMULA (in this time called SIMULA 67) were presented [1] to the world professional community at the IFIP Working Conference on Simulation Programmng Languages held in Oslo (Norway) in May 1967 [2].

Both the title of the programming language and the occasion of the presentation speak clearly on a relation between the language SIMULA and simulation. The relation led to a superstition that SIMULA is a simulation language. Although SIMULA is an excellent programming tool for constructing simulation programs it is not limited to simulation; in the seventies of XX century, it was sometimes called *universal programming language of the third generation*. When SmallTalk 80 was offered to the world programming community, being inspired by some properties of SIMULA, the term ***object-oriented programming*** (further ***OOP***) came into existence for SmallTalk 80, then for some other programming languages like it and then for any programming tool and/or technique that allowed to:

(OOP1) declare abstract concepts as structures of values and procedures,

(OOP2) use such a declaration Δ for generating any number (and in any sequencing) of computing objects carrying the values and procedures expressed in Δ,

(OOP3) use such a declaration Δ for any number of further declarations Σ, adding further values and procedures to those expressed in Δ. A declaration of Σ should behave similarly as that of Δ, i.e. (OOP2) and (OOP3) should be satisfied for it.

The abstract concepts of (OOP1) were called ***classes***, their ***specialisations*** in (OOP3) were called their ***subclasses***, the values of (OOP1) were called ***attributes***, the procedures of (OOP1) were called ***methods*** and the computing objects of (OOP2) were called ***instances*** of the given class, or – without their relation to any class – ***objects***. A subclass itself can be a base for formulating its own subclasses and so the process of specialization can continue and branch into trees of classes. The classes themselves are not able to make any computing; only their instances are able to do it, namely by ***sending messages*** to (other) instances. An instance *a* can send a message to an instance *b* so that the message carries a name *f* of a procedure (method) that *b* is able to perform. The message represents a demand like "instance *b*, perform procedure *f*". In the description of *f*, another messages can occur and so a message can activate an "avalanche" of message passing that could make the whole computing job or task. Now another principle of the object-oriented programming can be expressed:

(OOP4) a procedure (method) can be specified as ***virtual*** in a declaration of a class Δ so that its contents does not need to be yet defined but expected to be – may be in different versions – completed in subclasses of Δ.

Nowadays OOP is considered as something completely independent of computer simulation, but it is not true. Moreover, simulation caused that 40 years ago more than OOP was already discovered and soon implemented, which is sometimes called ***super-object-oriented programming*** (SOOP) [3], [4].

II. Start from Simulation Languages

Computer simulation is a method to study complex and complicated systems. It is the only method to study them in exact manner. The simulation model of a complex system is also a complex software product and simulation languages help to program such models. Their principle was, that their users do not describe what should happen in the computer during the simulation experiments: they describe directly the simulated system and the description is automatically translated into the computer code (in rare cases: automatically interpreted as running simulation model).

There is a large spectrum to view (and thus to describe)

Manuscript received May 4, 2007. This work was supported by the Grant Agency of Czech Republic, grant reference no. 201/060612, name "Computer Simulation of Anticipatory Systems".

Eugene Kindler is Professor of Applied Mathematics, now emeritus and as external specialists with the Department of Computer Science of Faculty of Science at Ostrava University, CZ – 701 03 Ostrava, Street 30. dubna no. 22, Czech Republic (phone: 420-220-801-945; fax: 420-221-914-123; e-mail: ekindler@centrum.cz).

complex systems. Already in the early stage of discrete event simulation, it was recognized that sophisticated behavior of the complex systems is cause as interaction of less sophistic behavior of ***processes***, which often follows instructions similar to computation algorithms but spread in time. Even a pair of two processes ruled by the same algorithms but spread into different time intervals may give a result that could seem chaotic. And thus it was already the first discrete event simulation language GPSS [5] that offered its users to describe the simulated systems as classes of elements owning analogous variables and governed by the same algorithms. Assignments, branchings, cycles and similar algorithmic substructures were allowed. Among them one could insert so called ***scheduling statements*** that interrupted performing the present algorithm and binding the continuing to some condition, like a certain value of the (simulated) time, an activating signal coming from another element that was just performing its algorithm, a free place at a facility or a storage etc. For the algorithms completing by scheduling statements and interpreted by every instance of a class, name ***life rules*** (of a class) came into use.

GPSS started developing the family of ***process-oriented simulation languages***. Although simple, it was used almost until the present days [6]. Through the next development over several languages, the process oriented simulation languages reached their top in language called SIMULA [7], [8] (after 1967 SIMULA I). Note that this language, implemented before 1965 was not the "true" object-oriented SIMULA. It was really a simulation language, did not allow subclasses, virtuality and procedures as components of classes, but

(S1) allowed declaring classes of objects with life rules based on a wide spectrum of almost all tools of ALGOL 60 (the most progressive algorithmic language of the sixties), and with attributes of a wide spectrum of types,

(S2) allowed generating instances of classes whenever during performing life rules,

(S3) allowed list processing and generating pseudorandom numbers by using standard functions and

(S4) permitted interactions between processes by so called ***connection statements*** that allowed interpreting some statements belonging to the life rules for a certain element *X* as concerning another element *Y* (in such a statement *X* could handle with the attributes of *Y*).

III. Simula and Hoare's Record Handling

The quality of the mentioned simulation language SIMULA stimulated the organizers of the NATO summer school on programming languages prepared for being in September 1966 in French Villard-de-Lans [9], to invite one of the authors of SIMULA, Ole-Johan Dahl, to take there a course on discrete event simulation languages [10]. At the Summer School, Dahl met another lecturer C. A. R. Hoare and his presentation on record handling [11]. Hoare proposed to handle with records (data structures) according to the following principles:

(RH1) there are declarations of classes of records, which have their name and which define the contents of every instance of a given class as a structure of components characterized by their names and types;

(RH2) among the components, also a ***reference*** type can be, specified in relation to the class to which it points;

(RH3) a class of records can be specialized to a subclass by adding further components;

(RH4) a class of records is open for generating any number of instances and for being base for specialization leading to any number of subclasses,

(RH5) if *A* points to an instance of a class that has a component called *B*, then *A.B* means "component *B* of *A*".

An example of such a class can be *person* introduced as having three components, namely *first_name*, *last_name* (of text type) and *date_of_birth*; the last component is of reference type, pointing to another class called *date* and introduced as having three components of integer type, called *day*, *month* and *year*. When *A* is defined as a name of an instance of class *person*, then *A.first_name*, *A.last_name* and *A.date_of_birth* and even *A.date_of_birth.day*, *A.date_of_birth.month* and *A.date_of_birth.year*, but expressions like *A.date_of_birth.first _name* or *A.year* are refused as illegal (senseless, inconsistent). Class *person* can be specialized e.g. to class *student* by adding further components like *school* (of reference type), *grade* etc.

According to Hoare, the records were passive elements and all handling with them had to be performed from a program algorithmized by means of traditional structures (therefore no processes were considered by Hoare).

O.-J. Dahl often reminisced that the Hoare's concept of record handling gave him a lot for their discovering the properties of the new SIMULA (see e.g. [12]). Really, the Hoare's ideas helped them making an efficient step in the development but the authors of the new – object-oriented and even super-object-oriented – SIMULA had to add many own discoveries. Firstly, let us concentrate to the aspects of the true synthesis of (S1)-(S4) with (RH1)-(RH5):

(DH1) class is a declaration composed of a set of attributes and life rules;

(DH2) a class is open for being a "pattern" of any number of its instances, i.e. for the elements that behave as Hoare's records if their attributes are taken into account, and as SIMULA processes if their "lives" are taken into account;

(DH3) among the types of the attributes, a reference type exists; when an attribute *A* of this type points to an element *E*, then in the life rules, the expression *A.G* represents "the attribute *G* of element *E*"; that expression is called ***remote identification*** or – more popularly – ***dot notation***;

(DH4) a specialization of a class *C* is a new class *D* with added attributes and life rules; there are rules how to join the added life rules to those belonging to *C*; *D* is called *subclass* of *C*, while *C* is called *superclass* (or – more frequently – prefix) of *D*;

(DH5) dot notation is not in contradiction with connection statements, both tools can exist abreast.

IV. Procedures as Components of Elements

A matter typical for OOP, namely the message passing, was not a consequence of the synthesis of the original Simula and Hoare's ideas. It was reached by including procedures into the contents of classes; namely, (DH1) was enriched to

(DN1) class is a declaration composed of attributes, procedures and life rules.

(DH2) should be enriched to

(DN2) a class is open for being a "pattern" of any number of its instances, i.e. for the elements that behave as Hoare's records if their attributes are taken into account, and as SIMULA processes if their "lives" are taken into account, and – independently on the state of such lives – they can perform any procedure occurring in the class declaration in case they are demanded to do it by any (other) instance.

To (DH3) another principle should be added, namely

(DN3) the dot notation can be applied also for procedures; if *X* points to an instance of a class that contains a procedure *P* with e.g. one parameter, then *X.P(a)* is a message demanding *X* to perform *P* with parameter *a*.

When the names of procedures are suitable formulated the messages may evoke (English) phrases (*subject.verb(object)*) or complex clauses (e.g. *member1.conjunction(member2)*).

(DH4) should be enriched in the following way to

(DN4) a specialization of a class *C* is a new class *D* with added attributes, procedures and life rules; there are rules how to join the added life rules …

V. Virtuality

Generating an instance of class *C* is represented by expression *new C*; already the original SIMULA allowed declaring classes with parameters; generating an instance of class *C* with one parameter *x* is represented by expression *new C(x)*. The parameters can be of the same types as the attributes. An idea to allow procedures as parameters easily arises. After a profound analysis of its consequences that idea was refused (the bad consequences are related to the block features of SIMULA – see section VII) and instead of allowing specifying procedures as parameters, the concept of virtuality was introduced:

(V1) in a declaration of a class *C*, a procedure can be specified as ***virtual***, in the sense that its contents is open to be defined (or re-defined) in any subclass of *C*.

Note: Virtual procedure is often represented as a certain "intelligent" aspect of OOP: when an element *x* obtains a message with a virtual procedure *P*, it decides on the way how to "understand" *P* and how to perform it: *x* has to overview its own "birth certificate" (i.e. to determine what class *G* occurred in the expression *new G* that generated *x*) and – based on *G* – it finds the relevant declaration of *P* and according to that declaration it performs *P*.

The virtuality became an integral tool of OOP (see principle (OOP4) in section 1) and the authors of so called radical OOP even demanded that every procedure should be virtual.

VI. Above the Level of OOP

While in simulation models one uses term ***element*** a general tendency in OOP exists to use term object (see section I).

Assume that the development of new SIMULA stopped at this phase. It would already reach the properties of an OOP language, as (OOP1) is covered by (DN1), (OOP2) by (DN2), (OOP3) by (DN4) and (OOP4) by (V1).

Naturally (DN1)-(DN5) represent more than OOP, because they contain life rules and scheduling statements.

But the life rules and scheduling statements immediately stimulated other features that were included and implemented in SIMULA. The first concerns virtual destinations in life rules. Note that the life rules do not need being performed exactly in the same order as they are formulated in the class declaration: transfers from one statement to another can be formulated so that the destination of the transfer can be given as a constant or as a results of "designational expression". Nevertheless, SIMULA allows using another possibility, too:

(V2) In a declaration of a class *C* a destination *D* of life rule transfer can be specified as virtual. Then *D* is expected to occur among the life rules formulated in subclasses of *C*. When the life of an instance meets the transfer to *D* it determines it according to its "birth certificate" (see the Note to principle (V1)).

Virtual destinations enable the life of an instance to be ruled in a sophisticated manner by switching life rules of any of its superclasses.

Another principle that leads beyond OOP concerns ***sequencing statements*** that are generalization of scheduling statements. A sequencing statement causes that the performing of life rules where it is met is interrupted and switched to performing life rules of another object. Every object has its reactivation point at that the first life rule that should be executed is stored. When the mentioned switch is performed from the life of *A* to that of *B* then the reactivation point of *A* is assigned to point to the statement *S* following the sequencing statement that was just performed. When sometimes after another sequencing statement causes a switch to *A* this object starts to execute its life rules from *S*. There are three sorts of the sequencing statements:

(SQ1) *call(A)*: when it is met in the performing the life rules of element *B* this performing is switched to *A*; at the same time, *A* puts into evidence that it was called from *B*; that takes effect in performing another sequencing statement, *detach*:

(SQ2) when the life of an object like *A* introduced in (SQ1) meets sequencing statement *detach*, the performing of its life rules is interrupted and returned to performing those of the object like *B*, i.e. of the object that "has called" *A*;

(SQ3) when the life of an object meets sequencing statement *resume(A)* the computing is switched to the life rules of *A* but no information on the object that caused performing the *resume* statement is stored.

By means of the sequencing statements, the scheduling statements used in simulation models can be programmed as

procedures; SIMULA offers them as standard ones.

VII. Block Structure of SIMULA

The mention section II, telling SIMULA to have used the large spectrum of algorithmic tools of ALGOL 60, contains also an information on the fact that ALGOL 60 introduced the complete block structuring. Block had local entities that were declared in its heading and that were for disposal only inside the block, while when the computing process left the block the local entities disappeared. The new SIMULA took that conception over and enlarged the concept of block in the following manner:

(BS1) *textual block* is a part of written source code composed of heading and operation part; heading contains declarations of local entities, among which those of variables, procedures and also classes can occur; when the computing process behaves like entering a block *B*, a *block instance* of *B* arises having the local entities as its proper ones; when the same textual block is so entered twice two corresponding block instances of it are generated.

Note that admitting the class declarations to be local in blocks enabled modeling several interpretations of the same concepts (interpretations e.g. following the same class declaration but influenced by their context, or interpretations following two different declarations of classes with the same names).

Class declaration is composed of class heading and class body; where class heading serves for introducing the class name, prefix, parameters, virtual entities etc., and the body contains the declarations of attributes and the life rules. There is a similarity between declarations of attributes and procedures (methods) in class body at one side and local entities in block at the other side. Therefore body of a class declaration appears similar to block and therefore the new SIMULA introduced the following principle:

(BS2) Class body is a statement (in practice it uses to be a block).

Combining (BS1) and (BS2) implied the following phenomenon

(BS3) Class body can contain declaration of other classes.

Therefore classes can be local not only in blocks but in other classes as well. If the body of class *C* contains declarations of other classes, then *C* is called ***main class***.

Main class can be an image of a formalized theory or of a concept of a generic model (model of a parametric system), possibly of a carrier of such a formalized theory (an expert) or of such a model (a modeler). In this manner, the new SIMULA admits introducing main classes for list processing and for event handling with respect to unique simulated time. SIMULA Standard [14], [15] offers such classes as standard tools (*SIMSET* for list processing and *SIMULATION* for scheduling processes with respect to a modeled Newtonian time flow), but other classes were programmed by SIMULA users.

Suppose *M* is a main class representing a model of a system *S* composed of objects, may be of processes. And suppose *C* is a class of processes figuring in the model, i.e. modeling components of *S*. Then *C* is a class local in *M*. It is possible to include a block *B* into the life rules of *C*, and it is possible that local classes *N1*, *N2*,… are declared in *B*. When the life of an instance *X* of *C* enters *B* a block instance of *B* is generated and *X* becomes and expert using notions *N1*, *N2*,… in his argumentations (or becomes a modeler using *N1*, *N2*,… for creating the model that he is to handle). When *X* leaves the block it is modeled like to lose his expert or modeling ability. When the lives of two instances of *C* are inside block *B* at the same time, for each of them his own instance of *B* arises and both of them can communicate like to discuss on the same more or less theoretical matter or like to communicate so that each of them has its own modeling computer to support his argumentation.

Some applications of special abilities of that super-object-oriented programming were mentioned e.g. in [16]-[19], where further references occur. They mainly concern nesting modeling, i.e. computer models of complex systems that contain modeling (imagining,…) elements.

VIII. Conclusion

As mentioned in section I, the new SIMULA has got a tag 67, in order to be distinguished from the old simulation language SIMULA that was then called SIMULA I. The former SIMULA I users turned to SIMULA 67 and therefore SIMULA I felt into oblivion. Thus in 1986, when SIMULA 67 was prepared to become an international standard under ISO, the tag 67 was abolished. The old SIMULA is really neglected.

Many popular OOP languages do not allow the life rules with scheduling statements for their switching and many of them permit classes only as global entities, because they do not consider block structuring. Note that the users of such languages (C++, SmallTalk, newer versions of Pascal,…) meet obstacles when they desire to program in “process-like” manner their simulation models.

Something like SOOP can be observed at JAVA and BETA [20]. Note that definition of JAVA syntax makes it not secure against some programmer’s errors that sometimes demand lengthy test during the program execution and sometimes can lead to the job collapse, while BETA uses syntax that is uncommon and distant from any programming usage. Both the languages mix the description of what should happen in the computer itself, with what is to be modeled. acknowledgment.

Acknowledgment

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

References

[1] O.-J. Dahl and K. Nygaard, “Class and subclass declarations”, in [2], pp. 159–174.

[2] J. N. Buxton, Ed., *Simulation Programming Languages – Proceedings of the IFIP Working Conference on Simulation Programming Languages.* Amsterdam: North-Holland, 1968.
[3] H. E. Islo, "SOOP Corner", *ASU Newsletter*, vol. 22, no. 2, pp. 22–26, May 1994.
[4] E. Kindler, "SIMULA and Super-Object-Oriented Programming", in [13], pp 165–182.
[5] G. Gordon, "A general purpose systems simulation program", in *Proceedings of 1961 East Joint Computing Conference*, New York: MacMillan, 1961, pp. 81–91.
[6] T. J. Schrieber, *An Introduction to Simulation Using GPSS/H™*. New York – Chichester – Brisbane – Toronto – Singapore: Wiley, 1990.
[7] O.-J. Dahl and Kristen Nygaard, "SIMULA – A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual". Norwegian Computing Center, Oslo, 1965.
[8] O.-J. Dahl and Kristen Nygaard, "SIMULA – an ALGOL-based Simulation Language," *Communications of the ACM*, vol. 9, pp. 671–678, September 1966.
[9] F. Genuys, Ed., *Programming Languages*. Academic Press, London – New York, 1968.
[10] O.-J. Dahl, "Discrete Event Simulation Languages," Norwegian Computing Center, Oslo, 1966. Reprinted in [9], 349–394.
[11] C. A. R. Hoare, "*Record Handling,"* in [9], pp. 291–346.
[12] O.-J. Dahl, "The Birth of Object Orientation: the Simula Languages," in *Software Pioneers: Contribution to Software Engineering*, M. Broy and E. Denert, Eds. Berlin: Springer, 2002. Reprinted in [13], pp. 15–25.
[13] O.Owe, S. Krogdal and T. Lychne, Eds., .*From Object-Orientation to Formal Methods*, [Lecture Notes in Computer Science, vol. 2635]. Berlin, Heidelberg, New York: Springer, 2004.
[14] O.-J. Dahl, B. Myhrhaug and K. Nygaard, "Common Base Language". Norwegian Computing Center, Oslo, 1968 (1st edition), 1972 (2nd edition), 1982 (3rd edition), 1984 (4th edition).
[15] *SIMULA Standard*. Simula a.s., Oslo, 1989.
[16] E. Kindler, "Object-Oriented Simulation of Simulating Anticipatory Systems", in *International Conference on Bioengineering Technology, Computer Science, Knowledge Mining, Prague, February 24-26, 2006*; *Computer Science*, C. Ardil, Ed, [Enformatika, Vol. 11], pp. 67–73.
[17] E. Kindler: "Object-Oriented Simulation of Simulating Anticipatory Systems," *International Journal of Computer Science*, vol. 1., 2006, no. 3, pp. 163–171.
[18] E. Kindler: "Agent-Based Simulation of Simulating Anticipatory Systens – Classification," in *14th International Enformatika Conference IEC 2006, August 25-27, 2006*, C. Ardil, Ed., [Enformatika, Vol. 14], pp. 1–6.
[19] E. Kindler: "Agent-Based Simulation of Simulating Anticipatory Systens – Classification," *IJIT – International Journal of Intelligent Technology*, vol. 1, 2006, no. 4, pp. 281–287.
[20] O. Madsen, B. Møller-Pedersen and K. Nygaard, *Object-Oriented Programming in the Beta Programming Language*. Harlow – Reading – Menlo Park: Addison Wesley, 1993

Eugene Kindler was born in Prague on May 22, 1935. He studied mathematics at Charles University in Prague (Czech Republic) during 1953-58. During 1961-6 he made his doctoral studies at the Research Institute of Mathematical Machines in Prague. At Charles University in Prague, he got grades RNDr (Rerum Naturalium Doctor, doctor of natural sciences) in mathematics and programming in 1968 and PhDr in logic in 1974, at the Czechoslovak Academy of Sciences he got grade CSc (Candidate of sciences) in physics and mathematics in 1970.

In 1958-1966 he worked with the Research Institute of Mathematical Machines in Prague, where he participated in designing new digital computers and then led the team for implementation of ALGOL 60 compilers for the designed computers. In 1967-1973 he worked with Faculty of General Medicine at Charles University in Prague, where he participated in projects in nuclear medicine and biophysics, and designed and implemented the first Czechoslovak simulation language. In 1974-2000 he worked with Faculty of Mathematics and Physics of the same Charles University, where he was concerned in education of undergraduate and doctoral students. For the courses, he developed mathematical theory of simulation modeling and introduced application of the object-oriented programming; the last activity soon passed the University domain over and focused into a working group at the Czechoslovak Cybernetic Society (one of the scientific societies existing under the Czechoslovak Academy of Sciences). Since 2000, he has worked with the new University of Ostrava, which was established in the 90-ies of the XX century after the change of political situation. And after the same change of political situation, he has got grade Professor in applied mathematics. He has collaborated with the same University as an external specialist since 2006, i.e. in the year when he reached the age of 71 and retired. As visiting professor, he worked one year with the University of Pisa (Italy, 1969-70) and one year with West Virginia University at Morgentown (USA, 1992-3). He worked three months with University of Blaise Pascal in Clermont-Ferrand (France, 1995) as invited professor and with the same University he worked six months as user of professor scholarship of French government (1997-8). Three times he was one month with the University of South Brittany in Lorient (France, 2003, 2004 and 2005) as invited professor. In 1983 he spent 3 months as foreign lecturer at Humboldt University in Berlin (Germany). During the years 1974-1989 he was responsible for projects of international collaboration between Charles University in Prague and Humboldt University in Berlin, and between Charles University and Latvian Governmental University in Riga (at this time Soviet Union, nowadays Latvia). Both the collaborations were oriented to computer simulation. He was the responsible person in the Czech Republic for works on two projects organized by European Commission and oriented to modernizing sea harbors with use of computer models (1995-2000). His Czech book on simulation languages, published in 1980, was translated in Russian and published in 1985 by the Moscow Energoatomizdat (Publishing House for atomic energy) and he participated with 100 pages in writing a book of 300 pages titled "Managing and Controlling Growing Harbour Terminals" and edited by The Society for Computer Simulation International in 1997 (ISBN 1-56555-113-3). Among seven hundereds of papers written by him in journals and conference proceedings, there is "Object-Oriented Representations of Formal Theories as Tools for Simulation of Anticipatory Systems" in *Computing Anticipatory Systems*. American Institute of Physics, Melville, New York, 2006 [AIP Conference Proceedings Volume 839], pp. 253-259 (ISBN 0-7354-0331-7). His present interest is object-oriented simulation of systems under intelligent control and its application in production, transport, ecology and service.

Prof. Kindler is a member of scientific or program committees of two Czech (MOSIS – Modeling and simulation of systems, annual conference, and ASIS – Advanced simulation of systems, annual conference) and two foreign (CASYS – computing anticipatory systems, biennial), ECMS – European conference on modeling and simulation, annual) periodical conferences on systems, modeling and simulation; besides, he is just nowadays a member of the International program committee of EUROSIM congress (Ljubljana, Slovenia, triennial). He is a member of committee of CSSS (Czech and Slovak Simulation Society). In his country, he got seven awards in mathematics, technology and medicine (beginning from the first prize in national Olympics in mathematics in 1953), in 1992 he was established senior member of The Society for Computer Simulation and in 2000 and 2005 he got best paper awards at conferences CASYS (Computing anticipatory Systems, Liege, Belgium).