Nested Models Implemented in Nested Theories

EUGENE KINDLER
Department of Informatics and Computers
University of Ostrava
30. dubna 22, CZ – 701 03 Ostrava
CZECH REPUBLIC
ekindler@centrum.cz

Abstract: The representation of knowledge by the object-oriented programming technique represents a tool for formulating theories that are as exact as those formulated by traditional ways of mathematics and formal logic. Moreover, this representation is suited to formalize knowledge systems on many material entities and science on them and automatically to invert the formulations into computer models. Striking is the fact that similar attempts existed already in the Aristotelian ontology and especially in his “hylemorphism”. Interesting factor is that the theories can be nested so that elements of one of them are formulated as carriers of (other) knowledge systems. Real applications that led to running computer models will be presented in the paper, together with specificiation of the essential properties of the object-oriented programming.

Key-Words: Object-oriented programming, Formal theories, Nesting systems, Computer models, Simulation.

1 Introduction – Aristotle’s Comeback?

1.1 Conventional and Algorithmic Abstraction
Using computers and – namely – communicating with them, one should be as exact as possible. The modern centuries gave our civilization a tool for exact communication, namely mathematics and – in its generalization – formal logic. Although there are many good fruits using that tool and intensively contributing to mathematization of life and social science, this tool does not appear too suitable for the communication with the automata, excepting proper mathematicians and some specialists in robotics, who prefer e.g. Prolog programming language.

There is a certain difference between two sorts of human thinking: on one hand the conventional thinking represented by pre-computer mathematics and formal logic, on the other hand algorithmic thinking. Contrary to the appearance that the algorithmic thinking is proper to the computer programming, it existed for centuries – instructions how to perform some time-consuming action under a priori unknown conditions are pregnant examples, even illustrating that “control statements” (presented often as something proper for the algorithms) – like cycles and branching – were used from of old by peoples without computers.

1.2 Algorithmization of Life Rules
The branch of discrete event system simulation carried so called process-oriented simulation languages [1] that enabled combining the mentioned “algorithmic” thinking (and abstraction) with describing simulated objects, i.e. real things or things viewed as possibly real in physical sense. The origin of the principle was as follows.

In order to facilitate error-less constructing computer simulation models, the simulationists were offered to describe the simulated system and not the algorithm used during the simulation model run: the description of the system was automatically translated into the mentioned algorithm. As the simulated systems often contain sets of elements that behave in a mutually similar way the process-oriented simulation languages enabled describing a “class” of such objects by presenting their similar properties called attributes (data carried by each of them) and their “life rules”, according to which each of them reacts when it is present in the described system. Just the life rules were formulated as algorithms, naturally completed by the mentioned “control statements” and so enabling different interpretation of the life rules for different situations in that different elements of the class were being situated. There are also so called scheduling statements occurring in the life rules; they enable automatic switching among the life rules performed by different elements so that parallel existence, life and dynamics of different elements can be modeled at a classical (single-processor) computer. The algorithm-like abstraction appeared quite natural for the simulationists, exact and secure against programming errors.

1.3 A Step to Philosophical Generalia
The introduction of “classes” into programming was the first step to introducing what the medieval philosophers called generalia and what is nowadays called notion or concept. The elements of the described system that were
formulated as belonging to a certain class are often called instances of the class. Inside the computer, every instance is represented as a data structure containing the attributes and the pointer to the expected (or just performed) life rule.

It is desirable that such a process-oriented simulation language is completely separated from the possibilities to describe in a direct way, what should be performed “inside” the computer. Such a programming language may be called pure. The described limitation is suitable for writing consistent descriptions (any programming error is detected independently of the used computer) and for portability of models among the computers equipped by software enabling to use the same programming language.

1.4 Hylemorphism in Programming

For such pure programming languages, an interesting property is valid: they enable to describe systems in that two or more elements can violate the axiom of the extensionality – these elements are considered as different even in case their attributes and their “life phases” are equal. Although the axiom of extensionality governs every mathematical branch it does not fully correspond to the reality. The pure process-oriented languages were the first exact tools that enable violating that axiom, surprisingly in a very natural way.

In fact, the violation of the axiom of extensionality is enabled by using different portion of the memory for the “inner” representation of every element. In case the considered language is pure, its user has no way to describe that memory portion; the portion behaves as materia prima (hylē) of the given element of the model and can be considered as a reflection of the material prima of the corresponding component of the described system. Similarly as materia prima in the Aristotelian philosophy, the memory portion is considered by any user as something that exists, that even carries a certain physical being of the element (and differing that being from that of another element that has otherwise the same properties), but that cannot be determined by some other properties. The contents of the memory portion may be viewed as what the philosophers called substantial form (morfē) but it does not seem to carry some interesting finding.

2 Yes, Aristotle’s Real Comeback

While the representation of the elements lead very near to what is called hylemorphism (Aristotelian teaching on matter and form), the development of the programing practice demonstrated that the attempt to the “generalia” introduced in the process-oriented simulation languages can be developed, enriched and led near both to the Aristotelian philosophy and to the “healthy thinking”. The development relates to “object-oriented programming” (further abbreviated to OOP), but that term should be explained in details, as development of OOP was rather complicated and its actual understanding as well.

2.1 Hierarchy of Concepts

As it was already mentioned in section 1.2, the labor for programming simulation models was facilitated by simulation programming languages by the following way: instead being forced to describe the algorithm applied during the simulation model run, their users were offered to describe the simulated system. In May 1967 a working conference on the simulation programming languages was organized and two authors Dahl and Nygaard presented there a paper [3]; its title Class and Subclass Declarations was so pregnant that nowadays one has to add a very simple explanation.

In a “normal” thinking (including scientific and technical intellection) one uses a hierarchy of concepts so that a concept B is understood as a “specialization” of concept A in sense that every entity that carries the essence of B carries the essence of A, too. In other words, the content of B is that of A upgraded of some properties and the extent of B is a subset of that of A. This relation among the concepts was transferred to that among their computer representation by classes: class C can be declared as a subclass (or a “specialization”) of class D so that new attributes and life rules are added to those of C. A consequence is that any instance of D can be viewed as that of C. And – naturally – another consequence is a simplification of declaration of the classes – e.g. for the mentioned class D, one is not forced to copy all life rules and attributes declared for C (Both the advantages correspond to those existing for the concept – for example, if one states that dog is mammal then he does not need to repeat the properties of the concept mammal, and he introduces that every (individual) dog is considered as (an individual) mammal.

The specialization of classes can be iterated: a subclass D of a class C can serve as a source for declaring a class E as a subclass of D etc. In other words, the technique presented in [3] enabled stepwise adding attributes and life rules, i.e. stepwise specialization (similarly as one considers e.g. concept dog as a specialization of concept mammal while mammal as a specialization of vertebrate). One class can be also specialized to several subclasses by adding different attributes and life rules (similarly as one can introduce that not only dogs but cats, cows, rabbits and mice are mammals as well). Note that C is called overclass of D and both C and D are called overclasses of E.

The possibility of introducing subclasses is a certain improving of what the classical philosophers introduced
by a pair *species* and *genus* and what e.g. Linné realized much later in biology.

### 2.2 Procedures, Functions, Methods

Another idea was presented already in [3], namely procedures, that were admitted also as components of classes. They have form of subprograms, may have parameters and are formulated as algorithms. Let such an algorithm be called “contents” of the given message. When a procedure $M$ is introduced as a component of a class $C$ a “message” can be sent to any instance $K$ of $C$. The form of such a message is in general a triplet $K\cdot M\cdot P$ where $P$ represents the parameters, $K$ is called *addressee* (of the message) and $P$ is sometimes called *selector*. An arbitrary element $E$ can send such a message that then becomes a request for a service: the addressee reacts by performing the contents of the selector, i.e. the given algorithm so that – excepting the parameters – the entities occurring in it are those of the addressee. Nevertheless, among these entities and the parameters, pointers to other elements can figure and using them the procedure can touch upon the contents of other elements.

The procedure can give a result. In such a case it is called function; in such a case, the requested service is often the result itself, but no limits are in general given to “side effects”.

A popular name for procedure is message. Note that the procedures do not strictly correspond to any philosophical notion but are of use in linguistic sense: if the name of a procedure is well devised a message $K\cdot M\cdot P$ can represent structures like name-conjunction-name, name-preposition-name, subject-verb-object, or like a mathematical expression where $M$ is a binary operator. The messages can be used as components of the life rules and of the contents of procedure.

### 2.3 Virtuality

The subclasses enable use of *virtual entities*: such an entity can be introduced for a class but its meaning is completed in subclasses of the class. Although such entities were already in [3] understood in a large spectrum (e.g. as targets of transfers in the life rules), in practice they are applied mainly in case of procedures: a virtual procedure is introduced only by its name (possibly completed by the parameters and by the type of the result in case it should exist), while its contents is declared in the subclasses.

If an instance $E$ of such a subclass $D$ behaves according to life rules introduced for its overclass $C$ and meets a message the selector of which is the virtual procedure mentioned above, $E$ turns to the declaration of $D$ and reacts to the message according to the contents of the procedure described there. In a complex system the user does not need to anticipate of which subclass of $C$ the element $E$ is an instance and so the message can represent a certain adaptive phenomenon built into $C$.

### 2.4 Object-Oriented Programming

During the development of programming, the possibilities formulated in [3] and several months later incorporated into programming language called SIMULA 67 [4], [5] were very slowly reflected over the world, until the development settled down in the end of the 80ies at a level called *object-oriented programming* (OOP). Other possibilities were introduced in [3] (and later described e.g. in [6]). We will return to them in section 4.

The essential tools of the OOP are subclasses, procedures and virtual procedures. In case a programming tool (language) does not offer some of these three tools the world professional community hesitates to view it as an OOP tool, even if its authors speak on the OOP.

Note that the life rules are not demanded as a component of OOP. The reason is in the fact that the ideas on the OOP aspects passed the frontiers of computer simulation over and were accepted by the remaining computer programmers; they degraded the life rules into “initializing” procedures: such a procedure is performed by any instance immediately after it is generated. Well, it can compute some attributes according to parameters (e.g. the circumference of a circle according to its radius that is given as a parameter), but for simulation such procedures are useless, as the languages themselves do not respect something like scheduling statements.

Popular OOP languages are Java, C++ and relatively new versions of SmallTalk and Pascal. None of them offers something like life rules with the scheduling statements. In case they are used for simulation the description of the simulated system has to be broken into “elementary transactions” [7] (called also events – see [3]). Another technique is that the switching among the statements is programmed with use of internal apparatus for switching among the computing tasks, i.e. intervene into what has no common with the simulated system. Java enable making it with use of the threads, the other mentioned OOP languages have to apply external procedures programmed at the assembler-like level. Both the techniques intensively violate the purity of the programming language (see section 1.3), as in case of switching among the life rules their user must rely on something that is related directly to the internal operation of the computer and without real affinity to the described (and modeled) system. The only OOP languages that allow formulating life rules and remain pure are Modsim III [8], BETA [9] and above mentioned Simula. To the last two ones we will return in the next sections.
3 Class Systems as Formal Theories

3.1 General Considerations
A class is an exact representation of knowledge, i.e. of something abstract. In general, more classes are used for a description of the same system,: for example, to a vehicle in a logistic system, a way (road,....) belongs together with crosses, places of stops and factors functioning at those stops; or to a patient in a hospital, physicians, rooms, surgeries etc. belong. A well designed OOP language is able to apply its compiler for a surveillance over the texts in the language formulated, and decide what is logically well founded and what is erroneous (e.g. a vehicle can move along a road but not along another vehicle, a physician cares for patients and not for an X-ray machine). Such a set of classes is bound by the “qualification” of attributes-pointers and should be in a certain way consistent and complete. Thus it represents a certain theory for which namely classes figure as axioms. Although the theory is exact it differs from theories occurring in conventional mathematics or formal logic: no quantifiers exist there but the texts are ready to be interpreted dynamically.

That was already one of the authors of the first OOP language SIMULA, namely O.-J. Dahl who noticed the fact that the programming languages can serve for representing concepts – he illustrated that at the plane geometry [10]: the center of a circle is a point, a point can be meaningfully tested whether it lies inside a circle, two points can be used for defining a straight line, for any closed curve it is meaningful to consider the area that it encompasses but computing the measure of that area is virtual and must be specially declared for various subclasses of the class of the closed curves...

A set of classes is a rather general theory, frequently describing some phenomena of the reality that one could characterize as branches, domains or fields, but is commonly used to construct computer models by means of describing the modeled system with use of the given theory (in terms common in the corresponding branch, domain or field). Such an application is fruitful in practice but does not seem to be interesting in general scientific view. Nevertheless, the initial development of the OOP contributed surprisingly much in this regard.

3.2 Local Classes
Already in 1967 (and presented in [3]) an idea arose (and was really implemented [4], [5]), to deal the declarations of classes similarly as those of the attributes or the procedures. In fact the idea was understood as follows.

The contents of a class is composed of attributes, procedures and life rules, where attributes are “copied” for every instance, life rules are interpreted individually for every instance and procedures are accessible for any instance that obtains a corresponding message. A class C can be added to the contents of another class H so that any instance of H can deal it similarly as it deals with its attributes: during its life (according to its life rules) or during performing a procedure, such an instance can form instances of C, and handle them and even introduce subclasses of C, their subclasses etc. and generate and handle their instances. But class C and its instances can become only “private” entities of a given instance of H. The consequence is that two instances of H behave as their class C would be two different classes that have no mutual relations. On the other hand, as a special case of a “usual” class, H can have its own attributes, procedures and life rules, too.

A class like H, i.e. a class that contains other classes, is called main class and the classes declared together with their attributes are called local or nested classes (or classes local or nested in the given main class).

In case more classes C, D, E,.... are local classes of a main class H its instances can handle them and their instances as forming an exact theory mentioned in the preceding subsection. One can observe that while the sets of classes mentioned in 3.1 were made systems by certain subjective and mental processes, if the members of such sets are nested as local classes into a main class they become a true system existing independently of any subjectivity and opened for “material” handling by the compiler of the applied OOP language. Any instance J of such a main class can be viewed as a carrier of a theory but the theory itself is private matter belonging only to J – in case the used OOP language is simple and well defined (i.e. logically consistent) a such a theory belonging to an instance of the main class cannot be shared by the other instances – the theories carried by different instances behave as completely different ones, contrary to that they are “copies” of the same sets of the local classes occuring in a common “pattern” (main class J).

One of the reasons of that “splendid isolation” is that the sharing of the theories among different instances of the main class could permit texts in the used OOP, which lead to undefined events (that could be very dangerous in case the used OOP language permits life rules – for any details see [11] or [12]).

The object-oriented programming that has use of local classes and life rules is sometimes called super-object-oriented programming.

3.3 Parametric Theories
Nevertheless the parameters of the main classes can be efficiently used in introducing parametric theories. As a simple example, plane geometry can serve. Let us explain that in details:
The main class geometry contains classes like point, line, circle, square etc., the contents (including their mutual interactions) of which similar as outlined in preceding subsection. It is known that the arithmetic with rounded real numbers is not exact and that a consequence is that it is not possible exactly to interpret the relations like “point P lies at a line L”, “the intersection of lines L and K coincides with the center of circle C” etc. Because of those obstacles it is suitable to introduce an attribute eps for the “whole geometry” that would be used for testing the equality of two points: they would be considered equal in case their distance is less than eps. eps is “universal” for any point figuring in the geometry and thus it has to be considered as an attribute just of the geometry and not that of some of its local classes. On the other hand, the values of eps carried by several different instances of the geometry can differ and so one may model containing those geometries in order to get complex information on their “accuracy”.

While the just mentioned example is very simple and may seem rather far-fetched, the same principle – with many parameters and enriched by dynamic behavior in time – was commercially applied for optimizing the parameters [13] of production, financial and biological systems [14].

### 3.4 Specializing of Theories

Another consequence of the fact that main class conserves the properties of any class is its specialization. If M is a main class and C is a class local in it then a subclass N of M may contain a class D that does not occur as local in M. D may be quite independent of the other classes local in M, but it is also possible that D is introduced as a subclass of e.g. class C. That is a logical consequence of the “inheritance” of the entities (like attributes and procedures) of M into its subclasses: the local class C is inherited as well and so it may be specialized. That possibility can be applied as follows.

M represents a certain theory that is a certain exact view to a certain component Q of the world, using concepts represented by the local classes. If N is a subclass of M it may concern a smaller component R of the world or a more detailed view on Q (or – combining both the offers – a more detailed view on R).

An example is a main class M oriented to a certain domain in designing steel plants in general while N is oriented to the same steel plants for that one knows more about their control. Then N represents a class where one governs more information on the same sort of steel plants. Nevertheless, N could be formulated in a different way, in fact to reflect only a special sort of the steel plants (limited in a geographical sense or in that related to special technologies, laws etc.).

### 3.5 Models of Individual Systems

The process of specializing main classes, introduced in 3.4 can be brought until description of a particular, individual system. Then the main class can be viewed as a formal theory oriented to a certain particular entity that occurs or is considered as possibly occurring in the world. Such a main class can be further specialized so that it describes the entity in a particular situation, possibly equipped by detecting tools that measure data arising at the entity and transmit them outside.

In such a case the main class represents a computer model of the described entity occurring in a specific situation (state, context, ...). So the main class can be stepwise specialized until relatively simple was of implementation of simulation (or other) computer models.

### 3.6 Carriers of Theories

If one models a real system S (or a system S that is considered as realizable, e.g. after deciding on its optimal structure), the idea of several instances of abstract theory may not be feasible in such a context. It is true but it is already the viewing to S (independently of its simulation), which helps: in the reality in that S could occur, the abstract theories are always carried by some “material” elements (possibly occurring in S as its components) – as example let us recite a thinking human H or a computer G, which may be present in S, perform some information processing P and according to its results send signals to other components of S in order to manage the run of S or at least to influence it. P may be based on certain concepts or even on a certain model constructed according to the concepts. In case of computer G, such a situation is that the process P is implemented with use of the theory (main class) carried by G. In case of human H, the situation is that the process P performed by H is based on some general concepts forming a certain theory: if one simulates a system containing such a human, the only choice is to formalize the human’s thinking as application of OOP, namely the theory for that the humans is its carrier.

The conclusion is that it is suitable to consider the main classes as concepts of (material) entities able to have (carry, own, use,...) an abstract theory. In practice it is better than consider a main class as a concept of abstract theory itself (although it is logically possible).

### 3.7 Examples of Applications

The optimizing computing processes mentioned at the end of 3.3 (13), [14]) can be viewed as models M of sessions of specialists E1, ..., En, each of which has a personal computer Ci for consecutive testing his hypotheses on the optimum. Every Ci is equipped by classes to concern the optimized matter, so the models of Ci (i=1, ..., n)
inside $M$ are carriers of a certain theory and differ by values of certain attributes that reflect $Ei$’s hypothesis of the optimum. These attributes can serve for a communication among the specialists so that any $Ei$ can read their values occurring at his “colleagues” and can change those of his own.

Other applications occur in simulation of transport processes where the carriers of the theories were drivers, sometimes replaced by automata (or tasks of a central computer) that managed automatically the drivers in order to use the shortest path and possibly a path secure against conflicts, local barriers or crashes (see e.g. [15], pp. 175-278, or [16]), and in simulation of industrial conveyors intelligently managed by a control computer able to anticipate possible suitable events (and have use of them) and improper ones (and avoid them) [17].

4 Nesting Formal Theories

In the last phrases the term anticipation was used. Note it is rather important notion often characterizing human thinking, free will and its essential influence on the indeterminism of the human society in case great number of humans apply it. The present section is applied to the computer modeling of anticipation in systems.

4.1 Nesting Main Classes

Some OOP languages (e.g. SIMULA) enable iterating of class nesting: e.g. main class $M$ may contain classes $A$, $B$, $C$, ... so that some of them, e.g. $C$ is itself a main class in that classes like $X$, $Y$, $Z$... occur as its local ones. In such a case, $C$ is a concept representing a “thinking” component the instance of that can occur among those of the other classes like $A$ and $B$ (and $C$ as well). In other words, $M$ is a carrier of a theory $T$ that uses concepts $A$, $B$, $C$ etc. where the concept $C$ represents a carrier of a theory $W$ that uses concepts $X$, $Y$, $Z$ etc. In such a case, $W$ is a theory nested in $T$, $T$ is a theory of systems in that carriers of theory $W$ can occur. In other words, $T$ is a theory of such systems that can contain “thinking” elements, namely those abstracting by means of concepts $X$, $Y$, $Z$...

The systems of that $M$ carries a theory are those applying theory $W$ to their own control and managing. Evidently the applications mentioned in 3.7. have to use such or similar structures of nesting theories.

4.2 Reflective Abilities

Suppose $W$ is similar to $T$, namely both the theories are represented by classes that are subclasses of a common overclass $U$ and contain similar local classes. In such a case what we identified $X$, $Y$ and $Z$ in 4.1 can be understood as certain modifications of classes $A$, $B$ and $C$. Namely, we can suppose $A$ and $X$ to be subclasses as a certain class $A^*$, $B$ and $Y$ to be subclasses as a certain class $B^*$, $C$ and $Z$ to be subclasses as a certain class $C^*$, etc., so that $A^*$, $B^*$, $C^*$ etc. are classes local in $U$.

In such a case, $M$ can be viewed as theory of systems that contain elements (instances of $C$) able to thing on themselves and on their environments. In other words, $M$ is a theory of systems able to reflect themselves at least at a certain level. Note that the pairs of similar classes (those that belong to the same overclass local in $U$, like e.g. $A$ and $X$) have to be really similar and not identical. Concerning pair $<C,Z>$ that is evident – otherwise $Z$ should be carrier of the same theory $W$ as $C$ and the chain of such carriers would be infinite. But in practice, also the other pairs $<K,L>$ do not contain identical classes – either $K$ should enable its instances to transfer their states to their “copies” belonging to $L$, or $L$ should enable its instances to “read” and accept the state of their “patterns” existing as instances of $K$.

The systems belonging under a class like $M$ described above, can be called reflective ones. The instances of a class like $C$ mentioned above may be called reflective elements; they often represent humans or (control) computers existing as components of such reflective systems. The class of reflective elements can be called reflective class. Note that a reflective system can have more than one reflective elements and those reflective elements can be instances of the same class or of different class. Thus a main class corresponding to a theory of reflective systems can have more reflective classes as its local ones. That can correspond to theories of systems with internal strain, battle, competition or rivalry, but also with internal stepwise development, cultivation and education (see e.g. an overview of running computer models in [18]).

4.3 Anticipatory Systems

A special case of reflective system is anticipatory one (such a term is used according to [19], according to [20] the author of which introduced a distinction between the anticipatory systems that have or do not have an explicitly visible reflective element, one should speak on anticipatory systems in week sense). Such systems use their reflective element for rather immediate anticipation of future consequences of an internal decision and for a possible modifying it, in order the consequences would be better for the system existence of at least for some objective of the system.

The main activity known at humans, which performs such anticipation, is a “rational imagination”, in other words an imagination of the future development related to the decision, nevertheless controlled by rational argumentation that eliminates the ideas that can be imagined
but not real. The latitude of anticipation itself and the ratio between the imagining and argumentation as well can vary for different situations. In case one wishes to model that activity in details at a computer the best (and may be unique) way is to map the human to a certain model of computer. In other words, the reflective element that is a human is mentally transformed as another reflective element of the same system and so it is described in the corresponding formal theory. The advantage of that trick is that what one makes mentally in the description is often made physically in case human’s anticipation is automated. In such a case, the human imagining is transformed to computer simulation and the rational corrections present at the imagining are transformed to more or less complex decision steps in the simulation.

Two different times are important in case of human anticipation – the anticipating human exists, lives and things in a certain time (that can be called public time) and during his own imagining another time in mapped in his mind, i.e. time to that he joins his imaginings (let it be called private time). In a rational imagining, the private time cannot be less than the public time but is often greater. Both the times should be in a certain relation, because the human’s intention is to interpret the results of his imagining into his real environment (in other words, the consequence of the rationalism of the imagining is that what was imagined as being in the private time should correspond to what could happen in the public time). Nevertheless, the difference between the public time and the real one can be illustrated at a sentence like “During one 30 seconds the human imagines what could come into being during the next two weeks in case he decides in a certain manner” – these 30 sec. concern the public time while those two weeks belong to the private time (the difference is especially striking in case the imagined state in the imagined two weeks tells the human not to accept the decision and to try formulating another one).

In case the human is transformed (“informatized”) to a computer the human’s imagining transformed to computer simulation is subject of very similar relations: contemporarily with the other components of the concerned system, the computer exists in their common public time, while during the simulation the computer handles a private time; nevertheless, if the computer work should be free of errors the events joined inside the computer to the private time should be interpretable as possible joined with public time.

### 4.4 Simulation of Anticipatory Systems

The domain of computer simulation develops in parallel with that of systems analysis and construction, namely with informatization of almost everything that belongs to the human society, and therefore of man-made systems, independently whether technical ones or social ones. The importance of anticipatory systems grows (the last crisis illustrated that phenomenon too clearly) and so among the simulated systems the anticipatory ones form a continually growing subset. When such an anticipatory system $S$ is simulated, its ability of anticipation cannot be neglected in the simulation model, otherwise the simulation would give false information (the proof is presented e.g. in [21] or [22]). The consequence is that a simulation model $M$ of $S$ has to contain a simulation model $\mu$ applied during the anticipation made inside $S$.

As $\mu$ reflects the private time and $M$ the public one, $M$ can be called public model of $S$ while $\mu$ its private one. Each of the models is described in a certain theory (see 3.5) – $M$ in theory $\Theta$ and $\mu$ in theory $\theta$. $\Theta$ be called public theory and $\theta$ private one. It is evident that both the theories have a lot of common components and that it is suitable to view them as specialization of a common “overtheory” $\Omega$. Note that among the components of $\Omega$, those concerning handling Newtonian time should occur. But such time exists in a lot of applications and so it would be suitable to have a certain overtheory of $\Sigma$, which would be staked-out to the knowledge of Newtonian time. A good OOP language should offer such a theory in a form of its “standard” main class; note that it has to relate to abstract, modeled time, not to real time of a computer (we know only on SIMULA [4], [5]). Such a main class represents a good discrete event simulation language and can be specialized to a large spectrum of simulation languages directed to certain domains (including languages for continuous or combined continuous discrete event simulation [23], [24]).

The reality is as follows: $S$ is a system and thus it is viewed as a part of reality segregated from the other reality. So it is reflected in theory $\Theta$ for any system like $S$ and thus for any model like $M$. But a part of $S$ (and thus of $M$) is $\mu$ that at one hand should reflect $S$ as a part of reality but at the other hand should communicate with its environment (or with the environment of its carries), because it should react to the state of the environment and it should answer by influencing a certain decision exiting in the environment. And that should be reflected in theory $\theta$.

The exact response is that theory $\theta$ should be nested in theory $\Theta$. $S$ the entities present in $\theta$ and handled by it exist in context of those present and handled in $\Theta$ and can communicate with them. Therefore one meets a case that a specialization $\theta$ of a theory (namely of $\Omega$ or $\Sigma$) is carried by an entity handled by another specialization $\Theta$ of the same theory.

In fact, that is common in the reality, where often a human (or a group of humans) things on him (them) own and on his (their) relations, but only OOP languages that allow class nesting enable simulating such entities. Inter-
esting is that that enable “nesting” times, namely that private inside that public. Note that it is to be recommended to use a pure language (as introduced in 1.3), otherwise the presence of two times may enable making programming errors that are hardly identifiable and repairable only with enormous effort.

In the other hand, a primitive offer of nesting theories may offer other fatal errors called transplantation [21], [25]. They can be illustrated by an analogy carried directly by the simulated system S: suppose S contains a queue Q which should carry some elements of S, and an element C that carries the private model μ. Among other μ should reflect the state of S and therefore Q too. A consequence is that μ has a queue q into that images j1, j2, … of elements J1, J2, … can enter. The person who describes S can make an error of transplantation e.g. so that instead writing “let j1 enter q” he writes “let j1 enter Q” or “let J1 enter q”. Such an error causes a start of mixing both the private model and the public one together and – if the error is not immediately discovered – the following manipulation with queues and times leads to a chaotic confusion that ordinarily ends with a computer fatal error (e.g. with non-existing address) so that the way to it from the first erroneous step cannot be reconstructed. Fortunately, Simula is equipped by a certain special logic that discovers transplantation error already as a syntax error, i.e. during the translation of the system description into machine code.

### 4.5 Examples of Application

Some application mentioned at the end of 3.7 can be viewed as simulation of anticipatory systems. Namely that referenced to [15] and [16] appears extremely interesting. If a transport tool has to move somewhere, it plans that activity in a cycle with two rather different steps: firstly it computes the shortest path to the destination (which is a certain work using one private theory o1) and then it simulates what could happen in case it applies this path (which is a work using another private theory o2). Note that o2 is much nearer to the public theory O used for the description of the whole system, than o1. If the simulation using o2 discovers that the application of the computed path would meet a problem like a barrier, a crash etc., at e certain place belonging to the path, this place is marked as denied for the path, and the computing pair is repeated; its first element offers a path different from the last one and the second element tests by simulation whether the new path is secure. And so the cycle is repeated until the first step computes a path that the following simulation step check up as secure. Note that in [15] only a model with the first step is described, while that with the both steps appeared in [16] and in greater details in [22] and [26].

Other applications concerning transport, production, health care and demography were referenced in [27].

### 5 Conclusion

During the first epochs of the human history, the people stepwise learned mutual communication. Their objective was to communicate as exactly as possible. After having conquered many obstacles, the human civilization came to the level of Aristotle and then to those who applied the Aristotle’s results (like patristic authors, Christian theologians of the first ecumenical councils and West European and Arabian scholastic scholars). The following steps came to communication on very complex systems and the scholars lost their ability of being exact – they either apply exact argumentation using inexact notions or even by inexact argumentation using inexact notions, where the inexact notions arose by more or less voluntary abstraction from complex reality.

What the civilization of the present decades has achieved in exact communication with the computers, may seem rather poor to the philosophers who know Aristotle’s philosophy. Nevertheless, if one judges the up to now development during maximum 60 years he has to evaluate it as a good period and can expect other results in future. Naturally, the real application of computing technique formulates and will formulate much more different questions than those common for philosophers. Note that even nowadays in computing there are some analogies of the Aristotelianism that are not applied directly in context of the object-oriented programming and therefore not presented in this paper, like analogies of the spectrum of reasons (causa materialis, causa finalis, causa efficientis causa formalis), related to ordinary differential equations, as mentioned in [20].

Combining certain aspects of early Hellenic philosophy, namely those reflected by the OOP, with enormous storage capacity and computing rate has already brought astonishing results, namely in modeling complex technological and biological systems. The aspects related to the super-object-oriented programming (see the end of 3.2) enrich not only OOP but the Aristotelianism as well and when they are combined with the computer abilities they will essentially contribute (and they have already done it) to a more exact research of technological and social systems that exist under and essential influence of human will (or under its “informatizationed” form in complex computer computations).

References:


