Abstract: Algorithmizing the control of a complex system needs to know it in details, especially when the control steps are tested at a simulation model of the system. Computing technique enables performing control so that possible consequences to more or less distant future states is tested and optimized. So the system becomes anticipatory one in the traditional sense. The simulation model of it consists of two (or more) levels – the “global” or “outer” one, in which one or more “internal” or “internal” models occur; they reflect the model used by the global one for reflecting the control existing in it. That phenomenon carries obstacles rooting in nesting world viewings, nesting formal theories and nesting parts of knowledge. Nevertheless, all the obstacles were already surmounted by use of programming languages that are object-oriented, process-oriented and block-oriented. In case such a language is isolated from the hardware aspects of the applied computer it can be used as a suitable (and may be unique) tool for exact description of complex systems, namely of those that are anticipatory and intelligent. In the paper, the development of such languages, their present state and simple examples of application in industry and services is described.

Key-Words: Complex systems formalism, Anticipatory systems, Sophisticated control, Object-oriented programming, Simulation, SIMULA

1 Introduction
Designing automatic control of a simple system was well supported by exact description of the controlled system and of the controlling block operation as well. That is a good experience existing for decades of years. The exact description was expressed using the tools of traditional (pre-computer) mathematics, like ordinary differential equations. A desire is to have something similar for description and control of complex systems. They differ from the “traditional” systems in several ways:
1.1. the number of elements is rather great;
1.2. although the elements can be grouped into “classes” of mutually similar ones, even the number of the classes is rather great;
1.3. moreover, the similarity of elements is often violated by some fine aspects and this imposes further requirements on the diversity of the classes;
1.4. the system input can be represented not only by numerical items but by a flow of entering elements;
1.5. the configuration of elements forming the system can vary during time; although that may be implied by 1.5, the variability of the configuration takes place often even in case the input is not a flow of elements;
1.6. the system inner state (including the configuration of its elements) can vary during non-equidistant events and so the flow of the entering of elements.
1.7. similarly as the input, also the system output can be formed by a flow of elements leaving the system (in general during non-equidistant events).

When a system is designed or when its control manner is modified, simulation of the system is a good way to yield information on the behavior of the system; especially, its reactions to various modes of control can be recognized by simulation and the optimal one of the considered ways can be chosen. So, a good (consistent, logically complete and legible) formalization of the studied system would be of great profit. Such a formalized description can serve as a non-confusing instrument of communication between experts studying the system and of “discovering what one does not yet know about the system”. In the past, a tool that enables such formalization was called specification language.

Already since the first attempts to digital simulation of systems at the beginning of the 60ies of the XX. centurty, a truth has been known that the best way to implement a simulation model at a digital computer is to have a compiler able to translate a legible description of the simulated system into computer internal program. In other words, such a way does not consist in “algorithmization in a narrow sense”, i.e. in description what should be passing inside the computer during simulation, but in description of the system itself, which should be independent of the applied computer. The fact that the algo-
rithmization is not necessary, would reduce not only the programming effort but also probability of programming errors, as the modeler is not forced to produce a mental transformation from the knowledge of the studied system to the simulating algorithm. From the other side, make real that conception, a compiler able to translate the descriptions of a certain rather great class of the systems into the machine code of their simulation models should exist. Note that to realize such a compiler is not a simple and cheap matter. Since the 60ies of the XX. century, the languages oriented to such descriptions and supported by the corresponding compilers have been called simulation languages.

The first simulation languages were limited to the domain of continuous systems that one could view as not too large analog computers (e.g. [1]). The difficulties with making compilers caused that for more complex systems the simulation languages differed from possible specification ones and forced their users to look for help in expressing many details that concerned the applied computer and had no interpretation in the simulated system (e.g. [2]). The unification of both the language types came in Simula I that was explicitly introduced as a “language for programming and description of discrete event systems” in five editions of its manual [3].

2 Discrete Event Simulation Languages

Simula I consummated the development of discrete event simulation languages, i.e. of the simulation languages offered to describe any system of the class of discrete event ones, i.e. systems that change their states in finite set of events so that the distances of them in time do not need to be equal. The consummating is to be understood so that the language was also specification one, that no object-oriented apparatus was included yet (see the next section) and that there were no special tools offered for legible description of inner models.

Simula I was based at the following viewing of systems:

2.1. The described system is composed of elements. Their number can vary so that an element present in the system may deliver another element that enters the system, and so that an element present in the system “disappears” in the sense that it loses all ways that can point to it.

2.2. The elements are viewed as data structures enriched by life rules. The life rules can be described by ordinary algorithmic tools common in algorithmic languages (for Simula I, the tools were taken from Algol 60 [4]) but so called scheduling statements could be included among them, denoting the duration of the give phase of “element’s life”, which has to elapse until the next step. When the life of an element meets such a statement its influence on the computation is interrupted and other elements may apply their life rules in the meanwhile. In the description of a system, the scheduling statements can be interpreted as “wait a certain (simulated) time period”, “wait until a certain signal comes”, or “wait until a condition is satisfied”. So the life rules enable to describe the dynamics of every element in an isolated manner and the scheduling of the lives of all elements present in the system is made automatically by the compiled program.

2.3. The items of the data structure of an element represent “properties” of the element (and are called attributes); they can vary in time. Their values can be of numeric, Boolean and textual type.

2.4. The elements with the same attributes and life rules can be viewed as instances of classes. The names and types of the attributes and the life rules belong to the class, the instances of which interpret that according to the particular state of their own life. The class concept enables describing large systems (those composed of enormous number of elements) in a rather short texts.

2.5. The interactions among the elements are facilitated by special standard routines oriented to list/queue processing.

The programming languages that offer scheduling statements are called process-oriented ones.

In Fig. 1 there is an illustration of the mentioned principles. It is a description of a very short and extremely simplified system composed of a source S, a sink R and

```
class pool(capacity); real capacity;
begin real size; ... end;

class vehicle(burden,source,sink,t_tr,t_l,
t_unl,sig_tr,sig_l,sig_unl);
real capacity, t_tr,t_l,t_unl,sig_tr,sig_l,
sig_unl; ref(pool)source,sink;
while true do
begin real carried;
inspect source do
begin if size=0 then passivate;
carried:=if size < burden
then size else burden;
hold(normal(t_l,sig_l));
size:=size-carried
end;
hold(normal(t_tr,sig_tr));
inspect sink do
begin if capacity-size < carried
then passivate;
hold(normal(t_unl,sig_unl));
size:=size+carried;
hold(normal(t_tr,sig_tr))
end;

S:=new pool(...); R:=new pool(...);
inspect S do contents:=volume;
for i:=1 step 1 until 5 do
activate new vehicle(..., S, R, ...);
```

Fig. 1
five vehicles that transport substance from S to R. S and R are “pools” and so each of them has a certain capacity and actual size of substance. At the beginning, S is completely filled by the substance, while R is empty. The times are subjects to normal distribution: the mean time for transport between R and S is t tr, that of loading at S is t l and that of unloading at R is t unl, the corresponding standard deviations are sig tr, sig l and sig unl. Actual size of the substance in S and R is called size. The vehicles start from S, they can load contemporarily and so they can unload. The end comes when S is empty or R cannot accept the coming substance. Note that in the simplification no conflicts are solved concerning the situation when several vehicles contemporarily take substance from S in case it is almost empty, and – similarly – when several vehicles put substance to R in case it is almost full.

An essential property that makes the applied simulation language a suitable tool for describing a system is that it is independent of the computing system. It is suitable to know that a possibility (or even necessity) to turn to another point (in the sense of graphical representation). So, if P and Q are “points”, i.e. instances of class point, one can apply e.g. P.over(Q) for getting the distance between P and Q. P.equal(Q) for testing whether P is geometrically equal to Q, and P.over(Q) for testing whether P is over Q.

3 Object-Oriented Programming

Since the end of the last millennium, object-oriented programming (in further text: OOP) is in common use, allowing formalization of complex concepts in form of classes as structures of data (attributes) and procedures (methods), and their stepwise specialization to much more complex concepts, namely by adding new data and new procedures, and by modifying contents of already specified (so called virtual) procedures.

Nevertheless, it was not later than in the second half of the 60ies of the last century, when OOP was invented (see namely [4]) and soon implemented in Simula 67 [5] (since 1986 called simply Simula [6]). And shortly after, one of the authors of Simula 67 demonstrated that the same language (or, better, its rather small part – see the next section) can serve as a tool for formulation of abstract concepts. That was presented in publication [7], explicitly titled in the mentioned way (Programming Languages as Tools for the Formulations of Concepts) and presented at a Scandinavian mathematical congress.

In Fig. 2, there are some illustrations following those in [7] and related to plane geometry. Class point reflects concept of point that has coordinates x and y, from which radius r and amplitude phi are computed in the life rules.

```plaintext
class point(x,y); real x,y;
begin real phi,r;
  real procedure to(K); ref(point)K;
to:=sqrt((x-K.x)**2)+(y-K.y)**2);
Boolean procedure equal(B); ref(point)B;
equal:=x=B.x and y=B.y;
Boolean procedure over(D); ref(point)D;
over:=x=D.x and y>D.y;
r:=sqrt(x**2+y**2);
if x>0 then phi:=arctg(y/x);
end;
point class material_point(mass); real mass;
begin real procedure potential_energy;
potential_energy:=9.80665*mass*y;
end;

class circle(center,radius);
  ref(point)center; real radius;
begin real circumference,area;
  Boolean procedure covers(K); ref(point)K;
covers:=radius>center.distance_to(K);
Boolean procedure conc_to(K); ref(circle)K;
conc_to:=center.equal(K.center);
circumference:=6.28418*radius;
area:=3.14159*radius**2
end;
```

Fig. 2

Further, every point is able to compute its distance to another point and perform relations testing whether it is equal to another point (in the sense of traditional geometry) and over another point (in the sense of graphical representation). So, if P and Q are “points”, i.e. instances of class point, one can apply e.g. P.to(Q) for getting the distance between P and Q, P.equal(Q) for testing whether P is geometrically equal to Q, and P.over(Q) for testing whether P is over Q.

Class point is then specialized to class material_point which is a concept governing all contents of class point and – moreover – attribute mass and procedure computing the potential energy. Further, class point is applied to express what is circle: its center is a point, according to the center and radius it is possible to test whether its array covers a point, and any circle A can test by A.conc_to(B) whether it is concentric to a circle B. The radius serves for computing the circle’s circumference and area – that is made according to the life rules.

Note that the example demonstrates that – differently from conventional mathematic expressing means – the programming languages are not limited by the extensibility principle: one can describe and model e.g. systems with more than one different points that have the same coordinates. Naturally, that is a good tool for manipulation with dynamic systems in that one of the geometrically equivalent points P may move and so change its coordinates, while the other stay at their place: so – in geometric sense – P becomes different from them contrary to that it was equal to them before. Evidently an inverse process is possible, too, during which – in geometrical sense – different points can join into one.
4 Agent-Oriented Programming

Geometry is a widely known exact domain with typical aspects of traditional mathematical abstraction and therefore it is a suitable subject for illustration presented in the preceding section. But geometry is oriented to static systems and in accord with that one can observe that the life rules occurring in the illustration are rather simple and without scheduling statements. The value of the illustration consists in that it is valid not only for Simula (to wit Simula 67) but for any OOP language: differently from Simula, popular OOP languages like SmallTalk 80, C++ and newer dialects of Pascal do not admit using the scheduling statements. Although the illustration presented in the preceding section can be adapted for them, they are not able express life rules of components of non-degenerated dynamic systems.

When the authors of Simula I developed Simula 67, they logically preserved the scheduling statements. Moreover, they extended them so that one can express switching life rules independent of time, too – e.g. for describing the strategic behavior (i.e. life rules) of different sorts of participants of a certain game. So the new Simula became not only an OOP language but also a process-oriented one. Synthesis of the object orientation with the process one results in agent-orientation.

Therefore the languages that follow this synthesis can serve for expressing exact concepts that are used in communication of dynamic systems, and may allow doing that in much more rich way than the discrete event simulation ones mentioned in section 2, i.e. the process-oriented languages that are not object-oriented. The object orientation allows exact introducing new language constructions reflecting verbs, conjunctions, prepositions etc. and use them for an exact description of systems, independently of whether the description is oriented to a person or to a computing system.

Note that OOP enables defining continuous processes in the systems and so describing and simulation of systems that are in general “combined discrete event continuous”. For that purpose, one can use statements like develop(t) of a form and meaning similar to “wait a certain (simulated) time period” mentioned un 2.2; but during the interruption before performing next life rules, the element continuously changes its attributes, respecting the system of differential equations described in a certain procedure equations. It was discovered not later than in 1975 [8]. Moreover, certain programming languages enable simulating continuous and combined systems containing elements developing according to differential systems of dynamically changing structure (including the number of equations): the first results were acquired in 1975 by use of simulation programming language NEDIS [9] that satisfied only a limited set of what is characteristic for the OOP; later they were adapted under OOP language Simula [10].

5 Systems with Complex Information Processing

5.1. Realism of such Systems

In this section, let us orient to the systems that contain one or more elements that perform complex information processing. Let such elements be called soft ones while the other ones be called hard ones. Under term complex information processing, let us consider information processing the description of that demands to use OOP. Such a demand can be a consequence either of the fact that the information processing is so complex that the describer of the whole system must turn to OOP for help if he wishes make the description well and just in time, or of the fact that the given soft element uses something like OOP by himself, independently of the viewing of the describer.

In Fig. 3 there is an illustration. S is the concerned system, just having 9 hard elements H1, ..., H9 and one software element σ1. All these elements are instances of classes C1, C2, etc. The soft element is just manipulating with a model m that is just composed of 7 elements e1, ..., e7, which are instances of classes d1, d2, ...

Fig. 3

The first question is how such systems are important. Really viewed, almost every system in that a person operates is of that sort, because such a person thinks and decides how react to the stimuli and possibly tries to optimize his action. Naturally, such systems were often simplified and idealized, so that the humans’ information processing be eliminated from their description (and models), but nowadays such a simplification appears as...
neither profitable nor necessary. Moreover, when one tries to transform the humans’ thinking to the operation of control computer, the demand of exact describing that operation becomes real. Thus the soft elements can be humans and computing systems (shortly: computers) and they can occur in social, production, transport, information, military and other systems. In addition, the complex information processing by a computing technique may arise independently on a deciding human pattern, and – as e.g. [11] shows – animals can also behave according to something that may be considered as their internal information processing (which could be important in describing and modeling e.g. ecological or agricultural systems).

The information processing performed by the soft elements often concerns the future consequences. In other words, at a certain time \( t \), the soft element should compute so that it gives a recommendation \( R \) what to do at time \( t \), but oriented (or, in a certain sense, optimized) to the future, at least to a time \( \tau = t + d \), where \( d > 0 \). The human performs that task when it imagines his own consecutive future acts where the \( i+1 \)-st act exists in a situation installed by the \( i \)-th act. Such a human transforms the system in that he is, to \textit{anticipatory} one [11], or – in a more refined classification – to \textit{anticipatory system in a weak sense} [12].

Although human performs such a rationally controlled imagining he is not able to do it in more steps and/or in a really exact way. The computing technique of the present time can substitute him in a much better way. What the computing technique performs is the true simulation. Thus, when one should describe/model a system in a really exact way. The computing technique of the present time can substitute him in a much better way. What the computing technique performs is the true simulation. Thus, when one should describe/model a system that contains not only hard elements but also one or more soft ones, he has to include the activity of the soft elements into the description/model. Let the model of the whole system be called \textit{global} one while the reflection of the activity of a soft elements be called \textit{internal} one. And – analogously – let the classes applied for description of the global model (e.g. \( C1, C2, \ldots \) in Fig. 3) be called \textit{global} ones while those applied for the description of the internal model (e.g. \( d1, d2, \ldots \) in Fig. 3) be called \textit{internal} ones. If both models are simulation ones, we meet \textit{nested simulation}. In case of an anticipatory system the internal model should reflect the state of the global one; in such a case one speaks on \textit{reflective simulation} [13]. It introduces special problems the solution of them will be described in subsection 5.3. At the present stage, let us note that they relate to the strong similarity of the global and internal classes.

5.2. Nesting Models

Two questions can arise. Firstly, is it really necessary to reflect the internal model \( m \) in the global one \( M \)? In other words: would not be suitable to neglect \( m \) in \( M \)? One should give a positive answer, the proof of which is as follows: if the internal model is neglected or replaced by something rather simple – may be \( \mu \) – two logical possibilities exist: \( M \) gives either false or good information on the modeled systems; the first case tells that the omitting of \( m \) was an error, the second case informs that the system \( S \) modeled by \( M \) was erroneously designed, because \( m \) can be removed from it or replaced by simple \( \mu \). Therefore when \( S \) is professionally designed and when \( M \) should give reliable information then \( m \) should be internal in \( M \).

Secondly, is it really necessary to nest \( m \) inside \( M \)? In other words, wouldn’t be possible to run \( M \) and \( m \) separately? For answering, one should be aware of the fact that any soft element has certain properties like the hard elements: it must perceive their properties in order to derive recommendations from them, it must transmit the computed recommendations to them and it exists in the same flow as they do. An illustration is in Fig. 4, where the arrows represent communication in the sense \( e1-H3, e1-H7, e5-H8 \) and \( \sigma1-H4 \) (e.g. \( e1, e5 \) and the soft element \( \sigma1 \) itself give some instructions to \( H3, H7, H8 \) and \( H4 \)) and in the inverse sense \( H1-\sigma1 \) (e.g. \( H1 \) sends a certain signal or information to the soft element). In especially complex systems, the soft elements may be generated, propagate and liquidated by hard elements. A consequence is that a model used by a soft element should be really internal in the global model. The most natural structuring would be as follows:

The simulated system \( S \) is composed of its elements, in general both soft ones and hard ones. The elements
follow their life rules and according to them they exist, develop and interact in a common Newtonian time flow. The life of any soft element can enter into an “information processing phase” during that it creates and runs an internal model. It applies it, so that it occasionally interacts with the other elements of S (in general with those hard and soft as well). When the “information processing phase” ends the internal model disappears and the soft element may wait or perform another function.

Translated into the description of a system and/or of its global model, the preceding paragraph could sound as follows: The global model is composed of the images of all the elements of the simulated system, i.e. of both the soft and hard elements. They exist in the common (simulated) Newtonian time flow in that they can “live” according to their own life rules. Also the image of any soft element exists in that manner. But in its life rules there are “information processing phases” during that they create internal models, run them and use them in more or less frequent contacts with the images of the other elements presented in the simulated system.

Therefore the natural way to describe a system – with a task to have a possibility of automatic translation of the description to the corresponding simulation model – is to describe the internal models inside the life rules of the soft elements. Otherwise, during the information processing phase, the description of the interaction of the soft element with the other elements would be difficult, incorrectly and being as a source of errors.

Note that when such a soft element is simulating, it itself exists in the Newtonian time concerning the other elements of the global model, but has relation to another Newtonian time, namely to that interpreted in the internal model. Both the time flows must be separated and secure against any mixing but in general they can interact, as it can be demonstrated by the following statement that can occur in the global model description: “While a soft element operates during time interval \( T_1, T_2 \) it simulates what could happen during time interval \( t_1, t_2 \); \( T_1 \) and \( T_2 \) concern the global model time, while \( t_1 \) and \( t_2 \) concern the internal model time – note that both the time axes may be expressed in mutually different scales and units, e.g. one time flow in days or years while the other one in seconds or microseconds, and the time starts (zero values of time) can differ as well.

5.3. Contribution of Block Orientation

At the beginning of the present section we expressed to be interested in the systems having internal models so complex that OOP technique has to be applied for their description. The consequence is that the information processing phases of the life rules of the soft element should be governed (and aided) by their special classes (reflecting the concepts used in description of internal models – see \( d_1, d_2, \ldots \) in Fig. 3) but – nonetheless – the soft elements should carry (and react to) some phenomena expressed and governed by the classes used for description of the simulated system (especially of its hard elements but possibly of some aspects of the relation of the soft elements to them – see \( C_1, C_2, \ldots \) in Fig. 3). In case of the anticipatory systems, the internal models are similar to global ones and forcing to use different terms (names of classes, of their instances, of the attributes as well as of the procedures) for the descriptions of both the models would be dummy.

In that situation, the block orientation introduced already in 1960 for Algol 60 language [14] is of use. In this language, the idea of blocks was introduced for algorithms, but it could be in an almost equivalent way formulated for life rules:

Life rules are composed of statements (like assignments, cycles, branchings, procedure calls and scheduling statements). They can operate over the attributes and with use of procedures of the “living” element, and – using “remote identifying” (often based on “dot notation” like in C++ or Simula) – with use of procedures governing by other elements over their own attributes. All these attributes and procedures are called global. Block is a section of the life rules enriched by local entities (in Algol 60: procedures and variables); the section is composed of certain statements that can operate not only over and with use of the global entities but over and with use the private ones, too. An illustration can be seen in Fig. 5, where 11 life rules of element \( E \) are represented by signs \( \rightarrow \) and their access to entities is represented by arrows in dashed line. All life rules have access to the attributes and procedures of \( E \), but let us that suppose the 5th - 8th of them are “bracketed” in a block; these four life rules have also access to the entities belonging to the block. When the life leaves the block these entities disappear.

Although block is a section of life rules, it figures as a special statement when considered as a component of the whole life rules structure. This implies that among the statements forming a block \( B \) a block \( b \) can occur. It is called subblock of \( B \). A statement forming \( b \) is also inside \( B \) and therefore the entities introduced for \( B \) become global (and therefore accessible) for \( b \). In this manner, the blocks can be recursively nested one into another up to any depth and the block with the “deepest” level of nesting can theoretically operate with all entities local and global in all the blocks that contain it. We wrote “theoretically”, as there is a certain obstacle that limits it; it is namely a “name conflict”, that occurs in case an entity local in a certain block has name that is equal to the name local in a subblock.
Algol 60 respected that rule completely: in case a block $B$ had got a local entity $x$ with the name $N$ equal to that of a local entity $y$ belonging to a subblock $b$, then $x$ was not accessible in $b$ under name $N$. If $x$ and $y$ were both declared as entities of the same sort (e.g. both were introduced as real variables) they figured really as two different entities, one being accessible inside $b$ and the other inside $B$ excepting inside $b$. In Fig. 5 that rule would be interpreted so that the arrow that leads out from the image of the block would not have its aim, because the arrow running inside the image should have priority.

Algol 60 had no classes (it was not an OOP language), but the block principle can be applied for the languages that are object-oriented and block-oriented as well (note that such a language must be logically agent-oriented, too). There are only a few languages that are so oriented and also block-oriented: Simula (exceptionally the first object oriented language – see e.g. [4]-[6]), Beta [15] and – when one turns both his blind eyes – Java. Although Java is rather popular nowadays, its object-orientation tools are poor and the language is bound with the hardware of the applied computer, which especially appears when one should formulate the scheduling statements. Beta has not the disadvantages of Java but its syntax is distant from many usances based on expression in mathematics and English and so it is not a suitable tool for describing anticipatory systems.

If the block orientation principles are applied also for classes then it is allowed to manipulate with the classes similarly as with the attributes, variables and procedures. Thus it is possible to introduce classes like $C1$, $C2$, … (see Fig. 3) in a block $B$ and those like $d1$, $d2$, … in its subblock, or even “deeper”: one of the classes introduced in block $B$ – namely that describing some soft elements – should have a subblock $b$ among its life rules, in which the internal model is described. In this subblock the classes (like $d1$, $d2$, …) used in the description of the internal model should be introduced. Block $B$ is the description of the whole system (and therefore of the global model). The way is very natural and corresponds to “normal” view of the systems.

In Fig. 6 a scheme can be seen that a bit corresponds to that at Fig. 3 (only the number of elements is reduced) – $H1$-$H4$ are hard elements of the described system, $\sigma$ is the soft element, all these elements are instances of classes $C1$, $C2$, … that are introduced in block $B$ which represents the described system and its model as well. The life rules are indicated like in Fig. 3. Among those of $\sigma$ block $b$ exist; when it is entered classes $d1$, $d2$, … are for disposal; elements $e1$-$e5$ are of their instances. For them an access to elements $H1$-$H4$ is possible, as well as for $\sigma$. Differently from Fig. 5, the arrows in dashed line represent relation “being an instance”.

When some names of classes, of their instances etc. used in $b$ are the same as that used in $B$, inside $b$ they are interpreted as concerning entities existing in $b$. That makes the description secure against the error called transplantation, when the describing person erroneously confuses both the models – e.g. declares to insert a real object (represented in $B$) into the image of a queue present as an instance of a class introduced in $b$.

6 Applications

Simula offers also other ways, that permit an access from $b$ to its environment even in case just described, and is even in such a case secure against transplantation, but such tools need rather detailed knowledge on that language (see e.g. [16]). Because of the same reasons, we only note that the same language has standard tools that allow having use of secure flow of simulated time in any block and thus to arrange two time flows as different as was mentioned at the end of subsection 5.2 is no obstacle. Let us present some applications.
6.1. Container Terminals

During the operation of the container terminals with ground moving internal transport tools, two questions have to be frequently answered:

1. A transport tool has to move to another position; what is the shortest path?
2. When the shortest path is computed at time $t$, its application needs a certain time interval $(t, t+d)$ where $d > 0$. During that time interval the configuration of places free of containers can change, especially in case more transport tools operate at the same time. Will the path computed in (1) acceptable until achieving the target?

The answers [17] were formulated so that the class of the transport tools was described as that of soft elements. Indeed, the life rules admit and realize the moves but two blocks $b_1$ and $b_2$ occur among them, $b_1$ for answering (1) and $b_2$ for answering (2).

The shortest path is computed by “pseudosimulation”, i.e. by a run of a simulation model of a fictitious system that would produce the same data as the desired routine [18]. For the shortest path, the fictitious system followed idea accredited to Dijkstra and Lee, namely that on a system composed of propagating pulses [19], [20]. Note that the pulses are instances of classes local to $b_1$ (like $d_1, d_2$... in Fig. 6), but the structure along that they move is composed of the real free places existing at the container yard, i.e. of instances of classes like $C_1, C_2$... in Fig. 6.

The answers to (2) were results of tests, what could happen when the path computed in $b_1$ would be used. That can be performed by true simulation: in block $b_2$, the classes concerning the transport tools are introduced and a simulation experiment starts, in which the image $w$ of the concerned transport tool $W$ applied the path computed in $b_1$. $w$ and the images of other transport tools are instances of a class local to $b_2$ (therefore, to something like $d_1$ in Fig. 6), that is homonymous with the corresponding class introduced for the whole model (i.e. not only that class has the same name but also its contents, namely the attributes and procedures).

In case the mentioned simulation experiment runs until $w$ achieves its target, the path computed in $b_1$ is accepted and assigned to $W$. In case the simulation experiment meets a conflict, a fictitious container is put at the place $p$ of the conflict and the life of $W$ returns to $b_1$ (in Fig 7 there is a scheme of the class to that $W$ belongs). That block computes a different path, as it cannot lead more over $p$, then test in $b_2$ is applied etc., until a path is positively tested in $b_2$, or until it is demonstrated that no secure path exists (nevertheless that variant was never met in modeling real cases). In Fig. 7, $d_1, d_2$... represent the classes related to the pulses and their propagation, $p_1$ and $p_2$ are symbolical outlines of the particular pulses, $C_1, C_2$... are some “copies” of the classes of transport tools occurring in block $B$ and $T_1$ and $T_2$ are simple outlines of the internal model images of the transport tools present in the system. Phase move represents the life rules describing the moving that the transport tool should perform after the last computed path is tested as secure. There are two returns forming cycles: the return delineated lower belongs to the cycle where questions (1) and (2) alternate for the same target, while the return delineated above is a part of the cycle where different targets alternate.

6.2. Circular Conveyors with Working Areas

Another sort of application concerns circular conveyors serving to transport “parcels” from outside to certain working areas, from working areas out and from one working area to another one. Every working area is linked to the “main cycle” of the conveyor through its input and output. In the working areas certain organization exists: parcels can wait there in a queue for being processed and – after being processed – they may wait in another queue if a danger exists that they could cause a crash when they immediately return to the main cycle of the conveyor. In Fig. 8, an example is presented, having five working areas, each of them organized so that both
the mentioned queues can accept only one parcel. The working areas are represented by segments parallel with the main cycle. Some parcels are outlined as boxes containing a triangle inside, the orientation of which – like an arrow – represents the sense of possible moves.

Many questions may arise during the operation of such conveyors. Present some of them:

1. A fault comes to the conveyor so that a working area is not accessible. Is it better immediately to interrupt all activities and to repair the faults, or let the conveyor work during some future time, in order to finish some phase of the production?

2. One should put a parcel that came to the conveyor, in order to let it transport to a certain working area. There are rather many of parcels at the conveyor and therefore it is possible that the working area will be occupied when the parcel comes to it and it is also possible that the parcel will perform the whole orbit at the main cycle, returning to the place where it was put to the conveyor from outside. In such a case, the parcel could obstruct other parcels in transport. What would be better – to install it immediately at the conveyor, or to wait some time? And if waiting is better, how long should it be?

3. In case there is a choice of several working areas for the processing of the parcel, which of them represents the optimal choice? (The instantaneous state, according that the decision is made, may change during the transport of the container to the chosen area.) That question may arise when a parcel comes to the conveyor (and so it may complicate the answering to question (2), or it may arise when a step of the processing is finished and another step has to follow, for which a choice of working areas exists.

4. The elaboration consists of several steps performed in general at different working areas. Some steps can be performed in arbitrary order. What is the optimal order related to the situation that is actual and that may change during the next time?

5. For some works, several different technological programs exist. What program is optimal in relation to the situation that can develop beginning from the present moment?

All these questions can be answered having use support of simulation. When testing the optimal investments, structure and basic control rules, the designed conveyor was simulated. Into its models, the simulation experimentation that was expected to help to decide according to (1) – (5) were nested [21], [22].

6.3. Other Examples
The first experiment concerning description and simulation of “intelligent” systems (still realized by old IBM 360 main frame computer) concerned a production hall equipped with automatic transport system performed by induction cars [23]. In substance, the methodology was similar as that described in 6.1 for container terminals, but the tests concerning conflicts were not built in. In other words, the structure of the description resembled that presented in Fig. 7 but block b2 and the inner cycle did not existed yet there.

A similar study was oriented to the mass public transport in regions [24]: currently the passengers who have to combine several bus lines imagine how variants of their choice could proceed; an idea is to offer transforming the individual and inexact imagining to simulation in service centers, which would be followed by recommending the optimal trace by phone. The same author then dedicated his interest to make models that would be used in consulting centers of regions and could advice the persons interested to domiciliate there, the expected demographic development (that of housing, education, employment market etc.). The models reflected that they will be used [25].

Another application starts with hospital models, under supposing that the everyday decision is supported by simulation for anticipation possible consequences of the instantaneous decisions [26]. That would enable designing long term control rules enabling flexible adaptation to purely human private desires.

Queuing systems controlled by one or more dispatchers applying simulation for recognizing possible consequences of their decisions (on shutting or opening the tellers behind the queues) were not only suitable demonstration models [27] but enabled being enlarged to model instructor with pupils, bureaucrats who decide slowly and without mutual communication [28] and even three-level model nesting: a model of a system of two competing queuing system that have their dispatchers using anticipation/simulation, but one of them anticipating/simulating what could anticipate the dispatcher of the adversary system [29].

7 Conclusion
The exact description of complex systems developed into a new branch, may be said into something that is near to abstract mathematics, that has use of algorithmizing, programming and systems analysis and that – vice versa – helps programming in a dialogue between human and computer, which is almost familiar but notwithstanding exact. The anticipatory systems in that either the human thinking or its computer models are built, support that development. In that domain, neither mathematics nor logic nor any other branch offers any exact communication tools. The programming languages that are object-oriented, process-oriented and block-oriented promise good way.
References:


