MODEL SYNTHESIS AND MODEL-ORIENTED PROGRAMMING – THE TECHNOLOGY OF DESIGN AND IMPLEMENTATION OF SIMULATION MODELS OF COMPLEX MULTICOMPONENT SYSTEMS

Yu.I. Brodsky

A formalization of the concept of a complex system simulation model is proposed, as a family of models-components with the standard organization of simulation calculations. This family is closed under the integration of models-components into the model-complex, so it is possible to synthesize fractal complexity models, without changing the computation organization. A new approach to the description, design and implementation of simulation models of complex systems arises – the model synthesis and model-oriented programming, allowing to exclude imperative programming, and to get the high degree of parallelism in the executable code.

Keywords: *simulation, complex multi-component systems, model synthesis, model-oriented programming, parallel and distributed computing.*

Introduction

The work is devoted to the description and simulation of complex systems, about which it is well known of what components they are made, what those components are able to do, what rules of interaction they obey. The challenging problem of modelling is to reproduce the behaviour and to evaluate the capabilities of such a system as a whole.

A new approach to the design and implementation of computer simulation models of complex multicomponent systems is introduced. It differs from the object-oriented approach. The central concept of this approach and at the same time, the basic building block for the construction of any more complex structures is the concept of the model-component. Model-component endowed with a more complicated structure than, for example, the object in the object-oriented analysis. This structure provides to the model-component an independent behaviour - the ability of a standard way to respond to standard requests of its internal and external environment. At the same time, the computer implementation of model-component's behaviour is invariant

with respect to the integration of models-components into complexes, which allows firstly to build a fractal model of any complexity and secondly to implement a computational process of such structures uniformly – by a single program. In addition, the proposed paradigm of the multi-component simulations allows to exclude imperative programming code and to generate code with a high degree of parallelism.

Analysis and synthesis – are the necessary stages of the design and implementation of complex systems and their simulation models. The analysis allows to decompose the complex system into set of components, simple enough for the subsequent research. With the help of synthesis, we are able to gather the whole system from these components. The processes of analysis and synthesis of complex systems and their models, are not formalized, i.e. they are more likely to be an art than a science. Nevertheless, there are number of tools designed to facilitate the analysis and synthesis of complex systems. One of the basic modern facilities is an object-oriented (OO) approach or object analysis [2, 13]. The results of the analysis of the system can be expressed by the class hierarchy in this approach. This hierarchy embodies the basic ideas of the system in the code of the methods. Then the synthesis of this system can be described using UML.

In our opinion, the problem here is that in the object analysis, there is no enough harmony between analysis and synthesis – the moment of the analysis greatly outweighs. As for the analysis in the OO approach, there the methods of decomposition of type "factorization" are developed, when the basic idea of the root class consistently specified by inheritance and overriding methods, until embodied in the code of the methods of the set of leaf classes generated from the root one. The trouble is that in the object analysis the decomposition into sub-objects is not developed.

The general theory of decomposition of mathematical objects - geometric theory of decomposition, developed by the school of correspondent member of RAS Yu.N. Pavlovsky [17, 18, 12], states that there are two main types of decomposition: F-decomposition – decomposition of factorization, and P-decomposition – decomposition into sub-objects. Any decomposition of any mathematical object has some mix of these two types of decomposition. From this point of view, the object analysis toolkit implemented in object-oriented programming languages in the form of inheritance and methods override is incomplete functionally: in presence of F-decomposition means, it lacks the means of P-decomposition.

In addition, in the known object-oriented high level programming languages there are no facilities for the synthesis of a software system from the leaf classes – it is left to the art of the developer. The means of the system synthesis describing from the objects appear in the UML [2, 13]. However, there is no consensus on the issue of how to build such a synthesis using UML. Range of opinions here extends from having to compile UML descriptions into high level programming languages, to the concept of model-driven software development, well-known in various versions of the model-driven concept of software development such as MDA (Model Driven Architecture), MDE (Model Driven Engineering) or MDD (Model Driven Development). Nevertheless, all these approaches offer no universal methods of the synthesis – the question remains in the area of arts.

The closeness hypothesis applied to complex multi-component systems

J.W. Goethe said in his Maxims and Reflections: "Man must persist in the belief that the incomprehensible is, in fact, comprehensible; else he would cease to do research". The closeness hypothesis in simulation is actually an expression of developer's confidence in the ability to create an adequate simulation model: to undertake its construction, you need to be sure of success. Let's try to discuss the possibilities of building computer models for complex multi-component systems.

The assumption of continuous dependence of the characteristics from the time is not natural for simulation models of complex systems. The discrete dependence often occurs. Furthermore, calculation of all the characteristics of a model is carried out by a computer as a rule. Note that implementing simulation calculations on a computer, we are able to process only a finite number of discontinuities of the first kind, during finite time. Hence, we seek the trajectory of our model $\vec{X}(t)$ in the class of piecewise smooth functions with respect to t.

We assume that the internal characteristics of the model depend on each other and on its external characteristics, and external characteristics do not depend on the internal. The closeness hypothesis postulates that the knowledge of the values of internal and external characteristics of the model at time t is sufficient to calculate its internal characteristics on some time interval $(t,t + \Delta t)$.

Further, we always put the discontinuity of the first kind in the beginning of the modelling step Δt, and then during Δt, we assume the model trajectory

$\vec{X}(t)$ to be smooth. We will calculate the model trajectory with a given accuracy $\varepsilon > 0$. For example, if the model trajectory is determined by the differential equation $\dot{X} = F(X, a)$, then for small enough $\tau > 0$ it is true:

$$X(t + \tau) \approx X(t) + F(X(t), a(t)\, \tau.$$

We now attempt to construct a trajectory of model $\vec{X}(t)$ on some macroscopic interval $[0, T]$ of model time, and along the way find out the assumptions that provide success for such a construction.

Definition 1.

We call a model closed at a point $t \in [0, T]$, if there is a number $\Delta t > 0$, , $t + \Delta t \in (0,T]$, which we call the segment of the model forecast for the point t, such that:

1. On the basis of internal and external characteristics of the model $\vec{X}(t)$ and $\vec{a}(t)$ can be determined whether there is a discontinuity point of the trajectory $\Delta\vec{X}(t)$ in t, and if there is – to calculate the gap.

2. Then, on the interval $(t,t + \Delta t]$, the model trajectory, starting from the point $\vec{X}(t) + \Delta\vec{X}(t)$, is a smooth function of time, for example, the solution of differential equations

$$\dot{X} = F_t(X,a).$$

Definition 2.

We call a model locally closed on the segment $[0, T]$, if it is closed at any point $t \in [0, T)$.

Definition 3.

We call the model predictable or Laplace on the segment $[0, T]$, if there is a finite partition of this segment by points $0 = t_0 < t_1) <,..., < t_n = T$, such that the model is closed at each of the points t_{i-1}, $i = 1,..., n$, and each of the intervals $(t_{i-1}, t_i]$, $i = 1,..., n$, belongs to the forecast segment for its left end.

Obviously, if the model is predictable in the sense of the latest definition, and we manage to find the appropriate partition of the segment $[0, T]$ by the points $0 = t_0 < t_1 <,..., < t_n = T$, then the task of building our model is solved. On the other hand, if conditions of the definition 2 are not executed, i.e., there exist points $t \in [0, T)$, where it is impossible to make even a small step $\Delta t > 0$, the task of the model building seems hopeless. So the request of the local closeness on the segment will be one of the basic requirements to the model.

A model predictable on a segment is called Laplace one, because the assumption about the possibility of constructing models on the segment is actually equivalent to what was called Laplace determinism in the history of science. That's how P.S. Laplace writes about this [16]:

«An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes».

Note that in the above citation, the past along with the future is also mentioned – and as will be seen further, the mention of the past is quite important.

Local closeness on the segment is not sufficient for predictability on this segment. A counterexample is so-called von Neumann's fly. Here is a slightly modified version of it, found in the task-book for primary school: two pedestrians go towards each other at a constant speeds. Between them flies a fly with a speed constant in absolute value and faster than the speed of any of pedestrians. Once the fly reaches one of the pedestrians, it turns and flies to the other. The question of the task is what the distance the fly will fly before the moment of pedestrians meeting.

Here all the history of the model is not sufficient to determine the speed of the fly at the time of the pedestrians meeting. Indeed, the speed of the fly is a discontinuous function of time, and at the time of the meeting of pedestrians has two limit values. So, at the time of meeting of pedestrians the old story of the fly's adventure is completely over, and the new could not begin without additional assumptions about its speed at this time.

Proposal 1.

Adding the requirement of left continuity of the model trajectory $\vec{X}(t)$ *at any point* $t \in (0, T]$ *can turn a model locally closed on the segment* $[0, T]$ *into one predictable on this segment.*

Indeed, let us move forward along the time axis from the origin 0, in accordance with the definition of the closeness at a point. If in a finite number of steps we do not reach the point T, it means that there is an accumulation point $\widetilde{t} \in (0, T)$. Because of the left continuity of the model trajectory, there is a left limit $\widetilde{X} = \lim X(t)$ at the accumulation point t. As we have chosen accuracy $\varepsilon > 0$ for simulation calculations, there exists $\widetilde{t} > \delta > 0$, such that

for all $t \in (\widetilde{t} - \delta, \widetilde{t})$, there is valid $\varepsilon > |\widetilde{X} - X(t)|$. We get to the point $\widetilde{t} - \delta$ in a finite number of steps, otherwise it rather than $\widetilde{t}$, would be an accumulation point. Let us set $X(t) = \widetilde{X}$ on the interval $(\widetilde{t} - \delta, \widetilde{t})$, and so we reach the point $\widetilde{t}$ in a finite number of steps. Then, in accordance with the fact that our model is locally closed on the interval $[0, T]$, we can continue our trajectory by a non-zero step, and here is a contradiction with our initial assumption that $\widetilde{t}$ is an accumulation point. ■

As we can see, the left continuity is sufficient for the model locally closed on the segment, to be predictable or Laplace on this segment. From the definitions of the closeness at a point and the predictability of the model on the segment, it also follows that the continuity of the left there is a necessary condition for predictability (possibly after redefining the trajectory of the model in a finite number of points).

The left continuity of the trajectory provides the conditionality of the current state of the model by some prehistory. P.S. Laplace said: «We may regard the present state of the universe as the effect of its past and the cause of its future» [16].

Note that the above reasoning is not only proclaims the opportunity to cut modelling for finite number of steps (assuming the left continuity of the trajectory), but gives a constructive way, how to do it. You have to start from the left end of the segment and "step over" the possible accumulation points – those segments where the trajectory of the system remains constant with the selected precision of modelling $\varepsilon > 0$.

Perhaps all the above considerations and suggestions seem too simple and even trivial. However, they imply some very important consequences for further discussion, which will be listed below:

1. As soon as we recognize the model characteristics functional dependence from the initial time t on the forecast segment $[t, t + \Delta t]$, as value of possible trajectory jump at the time t, either its subsequent smooth dynamics on $(t, t + \Delta t]$, – it is naturally to recognize that there is an ability to calculate these functional dependencies in the functional programming paradigm.

2. Functional dependence is unambiguous by definition. So, it follows that if for some reasons (for example, out of the model domain, or from the way of the computing process organization), it seems convenient to decompose this dependence on the number of concurrent computations, this decomposition must not violate this unambiguity mentioned. Thus, if two parallel processes try to modify the same characteristic of the model – it is just a bug of its de-

veloper – violation of the unambiguity of functional dependence (and not an objective reflection of the complexity of the model or the system simulated). Therefore, the calculations of the functional dependence decomposed into parallel processes are not equivalent to the original. Saving the unambiguity of calculations – is a necessary condition for such equivalence.

3. The Definition 3 of predictable or Laplace model implies that the process of computing of its trajectory naturally decomposes into alternating "jumps" and intervals of smooth dynamics. The model trajectory can be considered as left semi-continuous. It can have only a finite number of breaks, which happen instantly in the model time and depend only on the left limit of the model characteristics at the time of the break. At the smooth dynamics intervals the internal characteristics of the model are depend continuously on its characteristics on the left end of the interval and the time elapsed. This decomposition determines the rules of the simulation calculations organization, proposed in the next paragraph.

Further, we will essentially base on the above conclusions.

Fundamentals of model synthesis

Let us describe informally a concept of model-component – the basic concept of the model synthesis. The formal description of family of species of structure "model component" in the sense of N. Bourbaki [3] can be found in [4, 7].

The model-component we define must have internal and external characteristics – and in this aspect is similar to the object of the object analysis, which also has a set of characteristics. In addition to the characteristics, the object of the object analysis has methods – a set of skills. An object of object analysis has not its own behaviour – it is only the storage skills, from which, along with the skills of the other objects, the software system behaviour will be formed. The model-component, in contrast to object, is endowed with its own behaviour. In contrast to the object, you cannot call from outside model's skills. But like the computer operating system, the model-component is always ready to give a standard respond to any standard query from internal and external environment (which is just a standard combination of internal and external characteristics values of the model, because of the closeness hypothesis). In operating systems, several system services works constantly and parallel to each other, to enable standard queries servicing. In the model-component, several processes can developed in parallel. Each of these process-

es consists of alternating of methods-elements – elementary model's skills implemented as functions of model characteristics. The methods-elements switching is determined by the state of the model's internal and external characteristics, due to the closeness hypothesis. Next, we take a closer look at the rules of the elements switching.

One of peculiarities of complex systems simulation is the necessity of taking into account different time scales of the phenomena, so we will carry out the classification of the methods-elements of the model in relation to the model time:

1. Instant or fast elements. These are the elements, which occur instantly in relation to the model time. Instant elements – one of the ways to calculate obviously discrete characteristics of the model.

2. Distributed or slow elements. These elements have non-zero length in model time, that is, take a time not less than a characteristic for this model, average time interval. In addition, for such elements for every reasonable interval of the model time, the result of the execution of the element during this time interval matches. The distributed elements – are the most natural way of calculating of the continuous model characteristics.

If you remember the section devoted to the closeness hypothesis, we can say that the fast elements are focused on the calculation of the model trajectory jumps, and slow elements focused on the calculating of the trajectory smooth intervals.

Events control the switching of the elements in the process. Informally, events – are what you can not miss when modelling the dynamics of the system – synchronization points for its various functionalities provided by the processes. The points, where the stage of the model (values of characteristics received) requires corresponding respond from some processes of the model-component.

Formally, the event is a function of the model internal and external characteristics at the beginning of the simulation step. From the point of view of simulation calculations organization, the event is a method, the input of which is a subset of the internal and external characteristics, and the only output parameter – the predicted time before this event. If this forecast time is zero, it means that the event has already occurred.

For each ordered pair of the methods-elements $\{A, B\}$, if there is possible an switching between them in a process, a corresponding method-event $E_{\{A,B\}}$ is required, to predict the time of this switching. The events of kind $E_{\{A,A\}}$ are also

possible, where the element A is interrupted but not terminated, for example, if it doesn't need finishing, but calculated characteristics of the model-component that are important for the synchronization with other processes. The switching should be unambiguous. The simultaneous occurrence of the events $E_{\{A, B\}}$ and $E_{\{A, C\}}$ says only that the model developer in his design missed out the consideration of this case, which, perhaps, should correspond to the switching $\{A, D\}$.

So, let us sum up. The model-component, which is the main concept of the model analysis presented, is characterized by:

1. A set of internal and external characteristics.

2. A set of processes, each of them features the alternation of methods-elements, which implement different functionalities. Incoming parameters of elements are subsets of the internal and external characteristics of the model, their return parameters change subsets of the internal characteristics of the model.

3. Methods-events control the switching of the elements in the processes or interrupt the execution of the elements to synchronize them. Incoming event parameters are subsets of the internal and external characteristics of the model. If there is possible a switching from one element of the process to another – with such an ordered pair must be associated only one event. The event that interrupts the execution of the element with its further continuation is also possible.

The model-component is an elementary, but nevertheless, a full-fledged model of a complex system. Therefore, it can be run to perform simulation experiments, of course, if there are initial values of the internal characteristics and the method of observation of the external characteristics.

The terms of the model-component running are the following. First, it is believed that at the beginning of the simulation step current methods-elements of all processes and all of the internal characteristics of the model are known (at the first step - there are the initial values of the internal characteristics and the initial methods-elements of processes). Secondly, observability of the external characteristics at any moment is assumed. Further,

1. The events associated with the current elements of the processes are computed. The correlation of events with the current elements of the processes is determined by the rules of switching. The events can be computed in parallel, but to promote the computing process further, you should wait for completion of the computation of all the events. If there are events occurred, it is checked whether there are transitions to the fast elements. If there are any – the fast elements run (they become current). They can also be computed

in parallel, but to advance the computing process further, you should wait for completion of all the calculations of the fast elements, and then return to the beginning of the item 1. If there are no transitions to fast elements – the transitions to the new slow elements are made, and then return to the beginning of the item 1.

2. If there is no occurrence of the events, from all the forecasts of the events is selected the nearest $\Delta\tau$.

3. If the standard step of modelling Δt does not exceed the predicted time to the nearest event $\Delta t \leq \Delta\tau$, – we compute the current slow elements with the standard step Δt. Otherwise, we compute them with the step to the nearest predicted event $\Delta\tau$. Slow elements can also be computed in parallel, with expectation of completion of the latter.

4. Return to the beginning of the item 1.

All the events and elements are functions in the mathematical sense and in the sense of the functional programming paradigm, i.e. do not have stages nor side effects, that can give them the additional possibilities of parallelization when calculating.

In connection with these terms of the model-component execution, a question can be asked: what if the running of the fast elements in the item 1, as well as the decreasing of the time step in the item 3, lead to an infinite loop of the program, due to the emergence of system events accumulation points. Full guarantee, as was shown in the section devoted to the closeness hypothesis, can be given only for piecewise smooth left-continuous systems.

Still, we note that once we have demanded the unambiguity of simulation calculations of the elements and have achieved it – all running simultaneously in the model time elements can be calculated in parallel, on all available processor cores or distributed computers. If for some reason it was not achieved – the running support system has the full information about who and what changes (actually, it should update the relevant data in the database after execution of the elements) and it is obliged to issue the appropriate diagnose of the run-time error.

Models-components can be combined into the model-complex, and may be (optional) that some components explicitly model the external characteristics of some other components.

In order to describe the model-complex it is enough to specify:

1. What models-components and in what number of exemplars are included into it.

2. Commutation of the models-components inside the model-complex if it occurs, i.e., which the internal characteristics of which models-component, are the external characteristics of which components of the model-complex.

When combining components into the complex, it should be appreciated that the unambiguity of the computational process may be lost. This might happen if, for some reason, several components calculates the same characteristics of the modelled phenomenon. In this case, we can include into the model-complex a new component that as the external characteristics receives all the set of mentioned characteristics and as the internal characteristic somehow calculates the only value.

The model-complex consisting of many models-components, outside can manifest as a single model-component.

Let us introduce the following operation of combining the components into the complex:

1. Some new components may be included into the complex, for the unambiguity of the computational process.
2. The internal characteristics of the complex are the union of the internal characteristics of all its components.
3. The processes of the complex are the union of all the components processes.
4. The methods of the complex are the union of all the components methods.
5. The events of the complex are the union of all the components events.
6. The external characteristics of the complex are the union of the external characteristics of all its components, except all those characteristics that are explicitly modelled by any components of the complex.

This operation turns the model-complex into the model-component. This fact allows us to build the model as a fractal construction, the complexity of which (and accordingly the model detailed elaboration) is limited only by the desire of the developer.

Model-oriented programming

Historically, the changings of programming paradigms was accompanied by the consolidation and aggregation of the base instruments of the programmer's activity.

It all started with the machine instruction, then, with the advent of high-level languages – such a tool became an operator which implements some of the completed action, possibly with a few machine instructions.

The victory of structured programming ideas replaced individual operators and variables by standard constructions such as “cycle”, “branching”, sub-program-functions and data structures.

With the advent of object analysis, the object became the main unit of the design. It unites some kind of data structure with a set of methods necessary for the data processing. In addition, through the inheritance mechanism, you can build a hierarchy of object classes, developing, implementing and embodying basic ideas of the root classes of this hierarchy. This programming paradigm is currently the dominant and its basic concepts, such as class, object, data typing, inheritance, encapsulation, polymorphism is implemented with some nuances in most modern imperative programming languages, such as C++, Java, C#, Delphi and many others.

Model-oriented programming paradigm offers to increase once more the aggregation degree of the basic units of programming. It is proposed to design a software system from the model-component entities, which have its own behaviour in addition to characteristics and individual skills, i.e. are capable in any situation to give standard for them answers to the standard demands of the internal and external environment. Formally, this fact follows from the above-mentioned closeness hypotheses for the model.

Using the previously described invariance of the organization of simulation model computing process with respect to the integration of models-components into complexes, we can offer a new approach to programming – the model-oriented programming. This approach is proposed firstly, for the simulation models of complex systems, and, secondly, for the complex software systems (may be unrelated to the simulation), with a distinct multicomponent organization, where the synthesis of the whole from its parts is pertinent, and the nature and the ways of interaction among this parts are known in advance, so that they fulfilled the requirements of the closeness hypothesis.

Unlike the object of the object analysis, it is not necessary (nor even possible) to call methods of the model-component from outside. No need to care about the organization of functioning of the model-component. The model-component functions always (as always operates, for example, the computer operating system), in accordance with the rules described above, and always is ready to respond in the inherent in its design way, to the changes

happening inside and outside that are observable by its methods-events. So, we can completely forget about the internal structure of the once debugged model-component, using it in the future structures as a ready functional block just needed only to switch inputs and outputs correctly – and everything will work. Everything happens about the same as when we put a chip into a microassembly and then placing this microassembly on a circuit board. If at the each level of the project the commutation is reasonable – the chip will function properly in a much more sophisticated electronic device.

There were several approaches to object programming, where the objects had behaviour, in the history of the computer science development. These approaches originated among researchers, engaged in the problems of artificial intelligence, and are known as the actor model [14] and agent-oriented programming [19, 20]. However, although the author agrees with the main message of these approaches – the importance of modelling the behaviour of the agents of the system, – the details of this behaviour description in [14, 19, 20], in our opinion, is too related with the specific of the artificial intellect field, i.e. is not universal enough. This problem was discussed in details in [8].

Descriptions of models-components are declarative. Also declarative are descriptions of the unions of the models-components into models-complexes. A special declarative language LCCD (language of component and complexes descriptions) [7–9] is offered for these descriptions. A considerable share in this language have commutation operators.

Declarative descriptions of models-components on the LCCD determine their structure and functioning logic, thus completely separating them from the substance, as actions of models-components, carried out by the methods-elements, and the reasons for these actions identified by means of the methods-events.

Due to the closeness hypothesis, all methods of the model-component, as fast and slow methods-elements either methods-events, calculate some functions (in the mathematician, not programmer's understanding of the term "function") from some subsets of the internal and external characteristics of the model-component. Therefore, these methods may be implemented in the functional programming paradigm (which, incidentally, may increase the parallelism degree of the resulting code). The latter does not mean an appeal for transition to the lambda calculus – the functional paradigm may be feasible on all the favourites C, C++, C# and Java.

The result is that the imperative programming – the most stumble rock during the development and debugging of complex software systems [11, 21] – is not applied at all in the model-oriented programming. In addition, the implementation of even very complex software systems by the model-oriented programming methods is decomposed in a number of foreseeable declarative descriptions of models-components and models-complexes, and functional programs not dependent on these descriptions and on each other. Such a decomposition is very convenient for the collective development of large software systems.

It follows from the above-mentioned rules that determine the computational process of functioning of any model-component, that the more process has the model-component, the greater is the number of methods you can run in parallel. Because when uniting the models-components into the model-complex, the number of processes obtained is the sum of the processes of its components, it can be concluded that when the complicity of the model grows, the number of methods that allow parallel execution also increases. It may be increased even more because of the implementation of the methods of the functional programming paradigm. All this allows us to hope for the fruitful application of the model-oriented programming methods in high performance computing systems.

Model synthesis and object analysis

We now compare the fundamental concepts of the model-oriented and the object-oriented programming. First of all, note that the object-oriented approach is now presented in two forms: one that can be called “basic”, it is based on concepts such as class, object, typing, inheritance, encapsulation, polymorphism, which are implemented with some nuances in most modern imperative programming languages such as C++, Java, C#, Delphi and many others. The second, that may be called “advanced”, presented by unified modelling language UML [2, 13]. The UML includes all of the above basic concepts; in addition, its creators have opted for a sharp increase in the number of initial concepts and ideas. For example, apart from the “vertical” inheritance relationship that is often called a UML generalization relationship (the generalization relationship is directed to an ancestor from the child), there are relationships of association, composition, aggregation and dependence. Now it is possible to describe the behaviour of systems, even in several ways: interaction diagrams, state diagrams and activity diagrams. In the UML there

are many things possible to be done in a number of ways, which makes it a convenient tool for the professionals, but very difficult for the beginners. The authors of the language say: "UML is subject to the rule of 80/20, i.e., 80% of most problems can be solved using 20% of the UML" [2].

We continue the comparison of basic concepts. Such concepts as class, object, typing in the two approaches are understood near the same. It should only be noted that the roots of these concepts certainly are not to be found in C++ nor even in Simula-67, but rather in the works of N. Bourbaki [3], the structuralists of the twentieth century [1], up to the F. Klein's Erlangen program [15].

About the same refers to the characteristics and methods. As for the use of the methods – there is a significant difference. In the object-oriented programming the object is designed to call its methods from the various programs, and you can pass as arguments to the method and take from it any variables that match its signature – not necessarily the characteristics of the object. In the model-oriented programming the method of the model-component can work only with component's characteristics, and to call it "manually" there is neither possible nor necessary – it will be invoked automatically when required by the logic of the model-component behaviour. The encapsulation in the model-oriented programming does not allow direct access to methods. You can only operate with models-components.

This does not mean that the methods are not available. On the contrary, for example, in the realized layout of distributed simulation system [8, 9], a library of methods is published in the Internet for public use, and models-components are possible that do not have any local method. All of their methods can be physically located in different places of the Internet. However, you can only use the method included in a process of some model-component, and inside the model-component it will not work by itself nor by the outside commands, but only in the accordance with the logic of behaviour of its host model – such is the level of encapsulation. In some cases, it may seem more complicated, but this is the fee of no worries about the model-component behaviour organization – it always behaves as it uses to do, and so its inclusion in any complex is always just a matter of proper commutation.

Let us say a few words about the inheritance. Inheritance relationship for the set of classes of object-oriented programming language is a partial order. Classes that have no ancestors, but have descendants are called to them root or base. Classes that do not have descendants are called leaf.

Designing of a large software system with the object-oriented paradigm is laying the basic concepts and ideas of this system into the base classes of objects and then building a hierarchy of classes, developing, specifying and embody these ideas in a variety of leaf classes, with the help of which the target software system will be built. Such a deductive way of designing of a large software system is good when it is created "from scratch", as the world of Plato. In the simulations, however, the problems not in the creation of new worlds, but in the modelling of fragments of the existing one, are much more likely. In such a fragment, there can easily be gathered "…the pan, the divan, the basin, the box with three locks, the valise and a tiny Pekingese" (Popular in Russia children's poem by S. Marshak, translated by Richard Pevear). They do not appear to be deduced from the each other and to rise up to their common ancestors – is just pointless. For these tasks, "basic" version of the object approach lacks inheritance up from the bottom (in the UML such an inheritance is available). In a model-oriented programming integration of models-components to the model-complex can be considered as multiple inheritances up from the bottom.

Even if the object-oriented design has built the greatest hierarchy of classes using inheritance, still all the organization of the computational process lies on the developer of the system: for the system does something - the developer is to organize calling of right methods in the desired sequence. The most difficult stage of construction of the system is not formalized – it is an art.

The attempts to formalize the process of complex software systems designing gave birth to the UML. Apparently any system can be described with the help of the UML, and even from several points of view. The question is what to do next with such descriptions – there is no unity in opinions. Some specialists (for example, [13]) believe that the main value of UML is just in the application as a mean of recording and sharing formalized descriptions of the stages of the sketch and design of complex systems. However, there are a number of tools, which allows to compile the UML-descriptions into the billet classes of universal programming languages, and in this case we can speak about the mode of using UML as a programming language. However, here we remain within the object-oriented paradigm again – we obtain a hierarchy of classes and billet classes, but do not remove the need to write imperative programs to call in the correct order methods of these classes.

Conclusion

We tried to find out some necessary properties of the obtained models, basing on some generally accepted issues in the simulationists community, such as closeness hypothesis, unambiguity and deterministic simulation calculations, as well as the need of implementation of simulation calculations on the computer for the final time,. One such property is piecewise continuous, with no more than a finite number of discontinuities of the first kind, character model trajectory. However, overriding the trajectory models no more than a finite number of points, we can assume that the trajectory models of semi-continuous from the left.

Basing on some provisions, generally accepted in the community of simulation models developers, such as the closeness of the model requiring, unambiguity and deterministic simulation calculations, as well as the need of the simulation calculations implementation on the computer during the finite time, we tried to bring out some necessary conditions for the obtained models. One of these conditions is the requirement to the model trajectory to be piecewise continuous, with no more than a finite number of jumps. However, overriding the model trajectory no more than in a finite number of points, we can assume that the model trajectory is left semi-continuous.

The class of models selected, (namely, meeting at each point the closeness hypothesis, with piecewise smooth, left continuous trajectory) for which the local closeness hypotheses causes the successful model synthesis on a finite modelling time interval (Proposal 1).

The organization of simulation calculations proposed, oriented to the models with piecewise smooth trajectories, as an invariant with respect to the integration of models-components into the model-complex. This fact allows solving completely the problem of describing and synthesis of simulation models of complex multicomponent systems.

We can formally define a class of complex multicomponent systems simulation models, as a family of species of structure “model-component” in the sense of N. Bourbaki [3], and build on the basis of this definition a new concept of description, synthesis and implementation of simulation models – model synthesis and model-oriented programming, which is an alternative to the widely used object analysis and object-oriented programming, especially for the simulation of complex multicomponent systems.

The model synthesis is minimalistic in a set of basic concepts: it has the only basic concept – model-component and an auxiliary concept – model-complex, which after all, can also be a model-component.

The concepts of model synthesis and model-oriented programming exclude imperative programming from the simulation project – the most difficult in the design, implementation, and debugging. All the parts of the project are declarative and functional [11]. In addition, model-oriented programming produces a high degree of parallelism code.

The proposed concept of complex systems simulation has been realized in series of simulation models implemented under the influence of the model-oriented programming paradigm. For example, some episodes of Reagan's SDI functioning were simulated, the model of interaction of several countries was created. Also the tools for complex systems simulation were created: the system MISS [10], and the software for the workstation of peer-to-peer system of distributed simulation [8]. Currently, in the Department of Simulation systems and operations research of Dorodnicyn Computing Centre of RAS, we are working on implementing of the model-oriented programming for high performance computing.

The above concept of model synthesis is applicable primarily for synthesis of simulation models of complex multi-component systems. However, it is hoped that a similar approach can be used for the development of complex software systems, with the organization that fits for the paradigm described in the paragraph devoted to the closeness hypothesis, including the software systems focused on high-performance computing.

This work was supported by the Russian Foundation for the Basic Research, project 13-01-00499-a.

References

1. Blackburn S. *Oxford Dictionary of Philosophy*, 2 rev. ed. Oxford: Oxford University Press, 2008, 416 p.

2. Booch G., Rumbaugh J., Jacobson I. *UML. User Guide,* 2-ed., Addison-Wesley*,* 2005, 475 p.

3. Bourbaki N. *Elements of Mathematics, Theory of Set.* Springer, 2004, 414 p.

4. Brodsky Yu.I. Bourbaki's structure theory in the problem of multi-component simulation models synthesis *Innovative Information Technologies*, 2014, Part 2, Moscow, Prague: HSE. P. 234–244.

5. Brodsky Yu.I. Simulation Software *System Analysis and Modeling of Integrated World Systems,* 2009, V1, Oxford: EOLSS Publishers Co. Ltd. P. 287–298.

6. Brodsky Yu.I., Tokarev V.V. Fundamentals of simulation for complex systems. *System Analysis and Modeling of Integrated World Systems,* 2009, V1, Oxford: EOLSS Publishers Co. Ltd. P. 235–250.

7. Brodsky Yu.I. *Model'nyj sintez i model'no-orientirovannoe programmirovanie* [Model synthesis and model-oriented programming] Moscow: CC RAS, 2013. 142 p.

8. Brodsky Yu.I. *Raspredelennoe imitatsionnoe modelirovanie slozhnych sistem* [Distributed simulation of complex systems] Moscow: CC RAS, 2010, 156 p.

9. Brodsky Yu.I., Pavlovsky Yu.N. Razrabotka instrumental'noj sistemy raspredelennogo modelirovaniya [Developing an instrumental system for distributed simulation] *Informatsionnye technologii i vychislitel'nye sistemy* [Information Technologies and Computing Systems], 2009, N 4. P. 9–21.

10. Brodsky Yu.I., Lebedev V.Yu. *Instrumental'naya sistema imitatsionnogo modelirovaniya MISS* [Instrumental Simulation System MISS] Moscow: CC AS of the USSR, 1991, 180 p.

11 Brodsky Yu.I., Mjagkov A.N. Deklarativnoe i imperativnoe programmirovanie v imitatsionnom modelirovanii slozhnych mnogokomponentnych sistem [Declarative and imperative programming in simulation of complex multicomponent systems] *Inzhenernyj zhurnal: nauka i innovatsii* [Engineering journal: science and innovations], 2012, N2(2). P. 178–187.

12. Elkin V.I. *Reduktsiya nelinejnych upravlyaemych sistem. Dekompozitsiya i invariantnost' po vozmucheniyam* [Reduction of nonlinear controlled systems. Decomposition and invariance under disturbance]. Moscow: FAZIS, 2003. 207 p.

13. Fowler M. *UML Distilled,* 3-ed., Addison-Wesley, 2004, 192 p.

14. Hewitt C. Viewing Control Structures as Patterns of Passing Messages *Journal of Artificial Intelligence*, June 1977.

15. Klein F. *A comparative review of recent researches in geometry* Complete English Translation is here – http://arxiv.org/abs/0807.3161 (accessed June 1, 2014).

16. Laplace P.S. *A Philosophical Essay on Probabilities* translated into English from the original French 6th ed. by Truscott F.W. and Emory F.L., Dover Publications NY: 1951.

17. Pavlovsky Yu.N. *Geometricheskaya teoriya dekompozitsii i nekotorye ee prilozheniya* [A geometrical theory of decomposition and some its implementations], Moscow: CC RAS, 2011, 93 p.

18. Pavlovsky Yu.N., Smirnova T.G. *Vvedenie v geometricheskuyu teoriyu dekompozitsii* [Introduction into geometrical theory of decomposition], Moscow: FAZIS, CC RAS, 2006, 169 p.

19. Shoham Y. Agent-oriented programming //*Artificial Intelligence*, 1993, vol. 60. P. 51–92.

20. Shoham Y. *MULTIAGENT SYSTEMS: Algorithmic, Game-Theoretic, and Logical Foundations* Cambridge: Cambridge University Press, 2010, 532 p.

21. Zave P. A compositional approach to multiparadigm programming // *IEEE Software*, 1989, 6(5). P. 15–25.

Data about the author

Brodsky Yury Igorevich, Leading Research Scholar, Associate Professor
Dorodnicyn Computing Center of the Russian Academy of Sciences
40, Vavilov Str., Moscow, 119333, Russia
E-mail: yury_brodsky@mail.ru