INTERNATIONAL JOURNAL OF EDUCATION AND INFORMATION TECHNOLOGIES
DOI: 10.46300/9109.2021.15.1 Volume 15, 2021

Model-Oriented Programming as a Consequence of the Structural Theory of Multi-Component Complex Systems

Yury I. Brodsky
Federal Research Centre
"Computer Science and Control" of RAS,
Moscow Pedagogical State University
40, Vavilova street, Moscow, 119333
RUSSIAN FEDERATION

Leonid V. Kruglov
Moscow Pedagogical State University
14, Krasnoprudnaya street, Moscow, 107140
RUSSIAN FEDERATION

Received: January 25, 2021. Revised: February 25, 2021. Accepted: March 4, 2021. Published: March 10, 2021.

Abstract: - The paper offers a new programming paradigm, which implements CAD methods in programming, The level of encapsulation in model-oriented programming is higher than in the object-oriented approach. The key features of the MO-programming are declarative style (no imperative programs), and focusing on the distributed and high-performance calculations. The method proposed is based on the structural theory of multi-component complex systems, and is applicable for a rather wide class of tasks including the elaboration of simulation models of such systems.

Key-Words: - Education, Information, Complex Systems, Species of Structure, Model Synthesis, Model-Oriented Programming, Programming Paradigm

I. Introduction. Programming: from Machine Instructions to the Ontologies

Programming has evolved from machine instructions and data of three types (integer, float, and Boolean), all the while enlarging and complicating the object of the programmer's activity.

The first programming languages brought constructions such as operators, loops, data arrays. Until now, those who have enough of these tools, eg who solve mathematical physics equations on various grids, are ardent fans of the Fortran and are quite happy without being familiar with innovations such as classes, inheritance, or ontologies.

Structured programming has substantiated a set of basic constructs: sequence, selection, iteration, recursion, subroutine, block, sufficient for writing any program without using "goto" statement – one of the most popular in the previous period [1]. Data organization becomes more complex - the records combine different types of data. Another important conclusion of this period is that the coordination between the programming language and the computer hardware (eg, C and PDP-11) can significantly increase the efficiency of computations.

The next step of the programming evolution was the object analysis. The concept of a class defines the type of objects, which instances combine a data structure with the methods for processing this data. Also, through the inheritance mechanism, you can build a taxonomy of classes, developing and concretizing the basic ideas embedded in the root classes of this hierarchy. When the class hierarchy is built, the program is seen as a sequential activation of the desired objects and calling their appropriate methods. Here is the point, where the object analysis technology ends, and the informal art of the programming starts. So, we see that object analysis is not an end-to-end technology, as, for example, CAD technologies are. As for the analysis – yes, it undoubtedly presents, but as for the synthesis – it is not formalized. The synthesis is left to the art of the programmer. As for the data organization, this period was marked by the rapid development of database technologies, and any complex software system interacted with one or more of such databases.

On the verge of millennia, the programming domain became more and more complex. The interest increased in super-large forms – the worlds.

Even in entertainment, we know and love the worlds of Tolkien, Lewis, Efremov, Strugatskis, Star Wars, the Matrix, Harry Potter, Pirates of the Caribbean, Game of Thrones. A conceptual model is needed to deal with such a complex domain. In [2], it was said about the need to study complex systems in three worlds: material, informatics, and ideas (as Plato taught), and it was proposed to do this with the help of species of structure in the sense of N. Bourbaki [3]. In the programming practice of the current millennium beginning, this translates into the creation of ontologies for the domain of complex software systems. The term "ontology" is not bad and has already taken root in computer science, but philosophers are the first, who occupy and own it for a long time. To ignore this fact – is like teaching children to regard Beethoven as a St. Bernard dog. Nevertheless, in the introduction we will allow ourselves to use this term, preferring further "structure" or "conceptual model".

Ontology helps answer the question posed above: what to do after the ideas embedded in the base classes are expanded by inheritance into a coherent system of leaf ones. How not get lost in such a wealth (there can be tens of thousands of them, for example, .Net)? You cannot keep so much in your mind – it can explode – you involuntarily have to arm yourself with Plato's scheme of the three worlds. The ontology may turn out to be Ariadne's thread that can lead the programmer out of this labyrinth. Especially if it was worked out first, and only then the taxonomy of classes was created, basing on it.

The problem arises: we are aiming at more and more complex worlds in our activity – the complexity of ontologies of programming domains is growing. Is there any limit to this complexity? Modern programming is already suffocating with complexity: the imperative approach is complex; the complex structure of the software system and the connections between its components; data organization is complex; the behavior of the system components is complex: its logic is complex and the functionality of individual actions is complex. All this makes modern programming art that is practically inaccessible to an ordinary person.

On the other hand, the explosive development of information technology requires complex systems programming to become widely available, such as CAD technologies, which can be taught regularly in high school, and not art passed down from the Master to the chosen Padawans. The object approach, even reinforced by elements of conceptual modeling (construction of ontologies), is not such a technology – too many stages of a programmer's activity remain unformalized. The main tool is still imperative programming, which is extremely difficult to debug.

This article is intended to show a way out of the complexity impasse, if not for everyone in the world, but for a wide class of problems in creating complex software systems. The proposed technology implements CAD technologies in programming that have proven themselves, for example, in the design of microelectronics, where the processor contains tens of millions of transistors. The proposed technology does not use imperative programming at all. Besides, some programming solutions are proposed, that could significantly increase the computing performance when interacting with specially oriented hardware solutions. Corresponding experiments were carried out for a long time, but for some reason, they have not yet found wide application.

II. Elements of Structural Theory of Multi-Component Complex Systems

In Plato's dialogue "Parmenides", the distinguished Elean sophist Parmenides, his mature follower Zeno (known to us for his aporias), young Socrates, and even younger Aristotle (not the famous philosopher – he was not born yet, – but the future tyrant of 30) argued about the nature of this world – whether it is unit or plural. Since any of these statements is a great truth (by N. Bohr's or K. Gödel's classifications), its denial is a great truth also. So, the controversy lasted a long time, points of view were confirmed and refuted by a lot of arguments. Now philosophers regard the dialogue "Parmenides" as the source of all modern dialectics. This does not prevent some particularly cynical persons from believing that this was just an outdoor promotion campaign for Eleatic sophistry among the Athenians. Nevertheless, the formula: "One in all, and all in One" was and remains in use among the mystics of all times and peoples.

Since the modeling claims to reflect the real world, this formula is reflected in the theory of Model Synthesis and Model-Oriented Programming, which offers an end-to-end technology for description, synthesis, and software implementation of complex multicomponent systems models [4, 5]. The central point of this theory is the construction of

a universal agent for the agent simulation – the so-called "model-component" (really wanted to name it an object, and that name would be the most relevant, but alas, the term was already occupied and used in the object analysis, salute to Beethoven!).

A model-component is a formal mathematical object – a species of structure in N. Bourbaki sense. It is formally defined, for example, in [2, 4]. The family of models-components has two important features:

- The family of models-components is closed under the operation of uniting models-components into the model-complex. The complex received by an association of models-components is a model-component itself, and therefore, can be included in new complexes.
- The organization of simulation calculations is the same for all representatives of the family. This fact means a possibility of the universal computer program creation, which is capable to execute any simulation model if that is the mathematical object supplied with the species of structure from that family of models-components.

These properties of the models-components family permit to solve the problem of complex system synthesis from its agents. All kinds of imaginable agents are models-components and any unity of models-components is also a model-component. Thus, this theory can be considered as a mathematical model and interpretation (of course, on a more simplified and lower level, like any model) of the above phrase: "One in all and all in One".

The concept of Model Synthesis gives some consequences of a very wide application. We can match almost any agent complex system (physical, or technical, or social, or mixed) with its mathematical model (the model-component is a completely mathematical object of a certain species of structure), at least as a mental experiment. The class of systems was described exactly in [5], to which the Model Synthesis theory is applicable, and we will briefly repeat this description in subsection 3.1.

What does this give for solving the problems posed in the Introduction?

- When the complexity of the system domain grows, the complexity of its conceptual model remains qualitatively the same – it is a model-component always. The complexity of the conceptual model grows only quantitatively - it has more base sets, typical characterizations, and axioms, while its type as a species of structure remains the same.
- The consequence of this is that the complexity of the organization of the computation also does not change qualitatively, with an increase of the system complexity. Of cause, the number of calculations grows, but their type remains the same since the work is carried out with the same structure. This allows to develop and debug to shine a universal program for the computations organizing, capable of launching any object supplied with model-component species of structure.

Thus, it becomes possible to bypass the complexity trap when developing more and more complex software systems. There is no need to invent new and newer ontologies, as the subject areas change or become more complex. We need to learn how to see any of these areas as a single and unchanging model-component. And the Model Synthesis theory gives us step-by-step technology on how to find the same model-component in almost any subject area. Well, and the universal program for calculation organizing can work with it.

Thus, the uniformity of the conceptual model of any modeling domain, as well as the way of organizing its calculations, are the base of the Model Synthesis theory and Model-Oriented programming technology.

III. Elements of Model Synthesis and Model-Oriented Programming

The agent-based approach to modeling has been known since the days of Leucippus and Democritus, – contemporaries of Socrates and the Elean sophists. Since then, and to our days, many different types of agents have been invented for such modeling.

Let us try to show in the next subsection that the model-component is more than another heuristic attempt to come up with a new type of agent, – it is a logical consequence of the necessary conditions for the computer modeling possibility, first of all, of the closeness hypothesis. Let us try to get

acquainted with the closeness hypothesis in more detail.

A. The Closeness Hypothesis

What is the closeness hypothesis? The most important components of the model are its characteristics, which reflect the state of the modeled system and the state of the external world that affects this system. The first set of them is called internal, and the second - external.

The closeness hypothesis assumes that the knowledge of the internal characteristics $x(t)$ and external characteristics $a(t)$ of the model at the moment t is sufficient for a deterministic and unambiguous calculation of its internal characteristics over a certain time interval $(t, t+\Delta t)$ of positive length Δt. External characteristics $a(t)$ are considered observable at any time moment t. Here $x(t)$ and $a(t)$ are vector functions.

Unambiguity and determinism here refer specifically to the computation process. The subject area may be stochastic, but the calculator must definitely know when to turn on the random number generator and which probability distribution to use after.

The task of modeling is to predict the evolution of the system on the macroscopic time interval [0, T]. This means calculating the values of the system trajectory $x(t)$ on this segment. These calculations can only be performed by a computer if the system is complex enough, and this is the case we are talking about here. This means that in a finite time we can calculate the value of the system trajectory $x(t)$ only in a finite number of points $t_i \in [0,T], i = 1,...,n;\ t_1 < t_2 < ... < t_n$. We must be able to get an idea of the values of the trajectory at other points, based on the calculated ones, by linearly approximating the intermediate values, for example.

It follows from the above that we can construct only models with piecewise smooth trajectories. There will not be enough computer time for more complex objects. The possibility of a finite number of jump discontinuities follows from the fact that if the system is complex enough, then some processes will always occur instantaneously compared to the duration of the simulation step, at any reasonable step.

More about discontinuities: since we are modeling approximately, the modeling step Δt is considered so small that we distinguish only the presence or absence of a jump in the segment Δt, and do not distinguish where exactly it occurred, at the beginning, the middle, or the end of Δt. This allows us always refer the jump to the left end of the segment Δt, for definiteness. From the same considerations, we believe that at the modeling step Δt there can be only one jump (if there were more of them, we would still perceive them as merged into one). Thus, the presence of a discontinuity at a modeling step is similar to an ordinary event in the theory of event flows.

Now we can accurately formulate the closeness hypothesis for the complex systems under the study.

Definition 1.

We call a model closed at a point $t \in [0,T)$, if on the base of internal $x(t)$ and external $a(t)$ characteristics of the model

1. we can determine whether there is a jump of the trajectory $\Delta x(t)$ at t and if it is - to calculate it unambiguously;
2. we can calculate a number $\Delta t > 0$, such that the segment, which we call the forecast one $(t, t+\Delta t] \subseteq (0,T]$, and then
3. we can calculate unambiguously the model trajectory, starting from the point $\{t, x(t)+\Delta x(t)\}$, as a smooth function of time on the forecast segment $(t, t+\Delta t]$. ■

The idea of closeness at a point is illustrated in Fig. 1.

Important note: Fig. 1 also shows, that on item 4, we move the system time forward by Δt, and find ourselves at the beginning of the next step of the simulation, where can start with item 1 again.

Definition 2.

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

Local closeness seems to us to be a necessary condition for the possibility of building a model: it is difficult to imagine how to build it if a positive step forward is impossible from a certain moment. The requirement of local closeness on the segment $[0,T]$ covers each point $t \in [0,T)$ with an associated forecast segment $[t,t+\Delta t]$, on which we can calculate the trajectory of the system. It is an infinite cover. If we add the requirement of the left continuity of the system trajectory at any point $t \in (0,T]$, this will be enough to find its finite subcover (supertasks may arise, without this requirement, such as a von Neumann's fly or a Thomson's lamp) [5]. The possibility of the finite subcover choosing from the total forecast segments cover, means the existence theorem for the modeling task [5].

The existence theorem is undoubtedly important, but for this work, the most important is the definition of the closeness at a point and Fig. 1, illustrating it. What does that mean for the programming?

The fact is that no matter what the field of the modeling is (a technical system, the struggle of populations for survival, the epidemic spread, or social processes), the entire dynamics of the model is very simple and the same for any subject area. This is a four-stroke cycle shown in Fig. 1 and described in the definition of the closeness of the model at a point (with the addition of the model time transition to the next step), and similar to the Carnot cycle, or the work of the combustion engine.

B. Model-Component, Model-Complex, and Programming Language of Components and Complexes Descriptions (LCCD)

The model-component was formally described as a species of structure in the sense of N. Bourbaki in [2, 4]. We will not repeat this rather cumbersome description here but will quote it sometimes. The formal description of the component model as

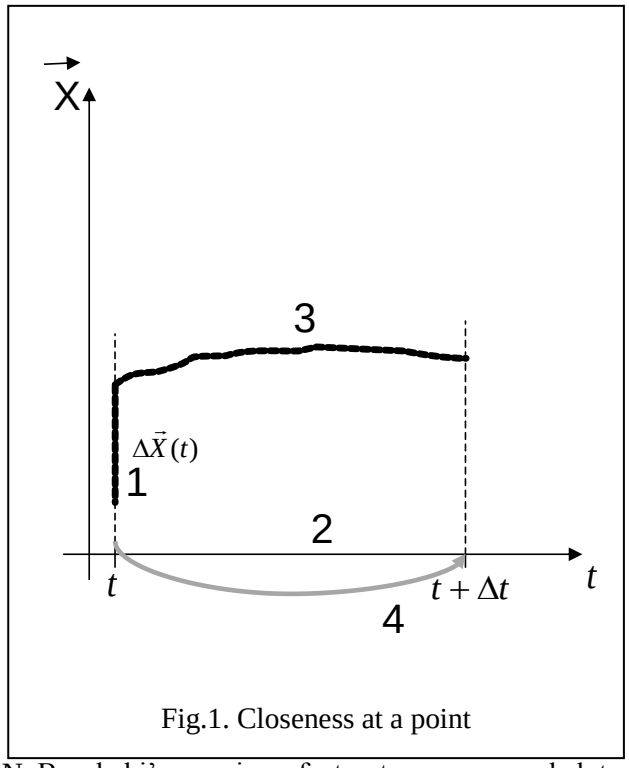

Fig.1. Closeness at a point

N. Bourbaki's species of structure was needed to prove that a complex that combines several model-components according to certain rules will be a model-component itself on the set theory language. Although a formal description is adequate and complete, it would be too unusual for a programmer to describe a complex system and its components. It is much more natural to use a special declarative programming language LCCD (the language of components and complexes descriptions).

A need arose for nonprocedural languages in which it is described not what it is necessary to execute, but who and how is arranged as well as who with whom and how is related. Some languages of a similar orientation with the names of their developers and sometimes the systems where they were applied are listed below:

- SQL (IBM, ANSI, databases) – 1986.
- MISS language (CC AS of the USSR) – 1986-1990, [7].
- IDL language (CORBA, OMG) – 1991.
- OMT (HLA, DMSO, IEEE) – 2000.
- Slice language (ICE, ZeroC) – 2003.
- XML (W3C, SOAP, Microsoft) – 1996-2005.
- UML (OMG, UML Partners) – 1997-2005.

Ontology languages such as Common logic, DOGMA, OntoUML, and many others are becoming more and more popular these days. Since

the trend is fashionable, many domains need conceptual modeling, and not everyone understands the meaning of such modeling, there are too many ontology languages now.

Since we assert the possibility of conceptual modeling of a wide class of domains with one universal structure - a model-component, we do not need a wide variety of languages. There is the only base concept in the Model Synthesis – a model-component and one auxiliary – a model-complex, which, upon closer examination, turns out to be also a model-component. Therefore, a minimalistic declarative language LCCD is proposed, which is a simplification of the MISS simulation system language, implemented in 1990, [7].

Fig.2. Von Neumann's fly

There are four types of descriptors in LCCD:

1. Data type descriptor.
2. Method descriptor.
3. Component descriptor.
4. Complex descriptor.

All descriptors, except a data type one; consist of heading and several paragraphs. A data type descriptor has only one paragraph. The data type descriptor is an optional construction.

All LCCD descriptors are compiled not into code, but database tables. Thus, there are three equivalent ways to describe the model-component: the species of structure, the LCCD descriptor, and the database tables. The first method is good for theorizing, the second is a developer's tool for describing the modeling domain, and the third is a conceptual model of the domain crystallized in a database so that a universal program for calculations organizing can work on this base. Next, we will try to bring all these three versions of the description in accordance.

Base sets: $X, M, E, \{M_j\}_{j=1}^{N}, \{E_j\}_{j=1}^{N}$. The easiest way is with X – the characteristics of the model-component. These are heterogeneous composite data types (like a **struct** in C language). The equivalent in the database is the same heterogeneous composite data, where a field of the appropriate size is reserved in the table for each data type.

M − the set of different implementations of methods-elements and E − the set of different implementations of methods-events, associated with the model. The correspondent construction in LCCD is the Method descriptor, although it is broader than just a description of the underlying base sets.

Let us illustrate how LCCD works by describing von Neumann's fly model. Two pedestrians walk towards each other at constant speeds. A fly flies between them at a constant absolute speed, greater than the absolute speed of any pedestrian. As soon as the fly reaches the pedestrian, it instantly turns around and flies to another, Fig. 2.

Below are the characteristics of the model in the database table.

TABLE OF CHARACTERISTICS

No	F_x	F_v	M0_x	M0_v	M1_x	M1_v
1	1	3	0	1	50	-1.5

The first line of this table is filled in manually. The subsequent ones are the simulation results. The data concerning the specific model should be supplemented with the simulation step Δt and the model time t, in the real database table. The row size did not allow us to add these variables to the table above.

The movement of the fly and both pedestrians (the calculation of the continuous evolution of the trajectory on the forecast segment) is provided by the slow method "move". It calculates the coordinate at the end of the forecast segment x' based on the value of the coordinate x and velocity v at the beginning of the simulation cycle and the duration of the forecast segment Δt by the formula $x' = x + v\Delta t$.

```
METHOD move;
//If the method type is not specified,
// SLOW is meant by default
ADDRESS: 192.168.1.75;
//where a realization locates
INPUT
double x, v;
OUTPUT
```

double x;
END;

The fly has a quick element “Uturn”. It always turns the fly around – changes the fly's speed v to $-v$.

METHOD Uturn: FAST;
// ADDRESS: local; by default
INPUT
double v;
/**********
As nothing is told about OUTPUT – it is the same by default, i.e.
OUTPUT
double v;
**********/
END;

The method-event “reaching” (the fly reaches the pedestrian) controls the transition from the slow method-element “move” to the fast “Uturn”. The reverse branch is unconditional, i.e. after "Uturn" always comes "move".

The "reaching" event algorithm is the most complex in this model. The fly must know the coordinates and speeds of both pedestrians. It finds a pedestrian at the speed of the opposite sign and divides the distance to him by the sum of the absolute values of its and that pedestrian's speeds. If the distance is zero, the event has occurred, if positive, – the time for the fly to reach the pedestrian is calculated.

METHOD reaching: EVENT;
ADDRESS: simul.ccas.ru;
INPUT
double x, v, m0x, m0v, m1x, m1v;
// OUTPUT of all the methods-events always –
// double dt; – forecast of its occurrence time.
END;

The main result of compiling the example descriptors is to populate the following table, the rows of which indicate where to look for implementations of the methods it contains.

TABLE OF REALIZATIONS

No	Method	Assembly	Address
1	move	Man	192.168.1.75
2	Uturn	Fly	local
3	reaching	Fly	simul.ccas.ru

We see that not all the information specified in the descriptors is reflected in the table. It will be reflected in other tables. The constructs of the species of structure, the LCCD language, and the database do not correspond bijectively to each other. Each of these methods has its logic. The important thing is that there are three ways to describe completely the model-component.

The "Von Neumann’s fly" model-complex consists of two instances of the “Man" model-component and one instance of the “Fly” model-component. The “Man" model-component has a single process consisting of a single slow “move" method. The "Fly" model-component has a single process consisting of a slow “move" method, a fast “Uturn" method, and a “reaching” event that controls methods’ switching. Therefore, the model-complex has three processes: two got from the instances of the “Man” and one – from the “Fly”.

TABLE OF METHODS AND EVENTS

No	Method / Event name	Type	Current	Process	Realization
1	Man_0_move	1	☑	1	1
2	Man_1_move	1	☑	2	1
3	Fly_0_move	1	☑	3	1
4	Fly_0_Uturn	2	☐	3	2
5	Fly_0_reaching	3	☐	3	3

We continue our review of the base sets correspondences. $\{M_j\}_{j=1}^{N}$ and $\{E_j\}_{j=1}^{N}$ are methods and events of processes. The table above is just about the methods and events of the processes. Besides, it contains some information from the method descriptors that are not included in the realizations table. It also includes the information on initial methods of processes, which is given by the typical characterizations $\{m_j^0 \subset M_j\}_{j=1}^{N}$ of the species of structure “model-component” and in the LCCD language, these methods are specified in the elements paragraph of the component descriptor.

The first column of the table is one of the keys. The second contains the names of methods and events. They are quite complex since they are obtained automatically when the complex descriptor is compiled into the corresponding component descriptor. The third column is the element type: 1 – slow method; 2 – fast one; 3 – event. The fourth column contains Boolean variables, which indicate whether this method is current. At the very first simulation step, the initial methods are specified. The fifth column indicates to which process the element belongs. The last sixth column is the

secondary keys one, which indicates the item's position in the realizations table.

TABLE OF SWITCHES

No	Process	Current method	Next method	Event
1	3	3	4	5
2	3	4	3	0

The switch table sets the correspondence between the methods to be switched and the events. There is only one event, and the reverse transition is unconditional, which is shown in the table by setting the impossible value to the secondary key. In columns from three through five, there are secondary keys that indicate the position of the element in the table of methods and events.

In the model-component species of structure, the analog of this table are typical characterizations $\left\{sw_j \subset E_j \times M_j \times M_j\right\}_{j=1}^{N}$ with rather cumbersome axioms R_8. In the LCCD language – the switches paragraph in the component descriptor.

Here are some examples of the component descriptors.

```
COMPONENT Man;
PHASE
double x;
PARAMETERS
double v;
ELEMENTS
 move;
 COMMUTATION
 move.x = x;
 move.v = v;
 x = move.x;
 END;
```

We can see that the characteristics of the components are divided into internal (PHASE) and external (PARAMETERS) in the LCCD. This is done for additional control during the compilation of descriptors: for example, external variables cannot be on the left side of the commutation statement of the parameters returned by methods.

```
COMPONENT Fly;
 PHASE
 FlyPhase:
 double x, v;
 PARAMETERS
 FlyParam:
 FlyPhase man0Phase, man1Phase;
 ELEMENTS
 move, Uturn;
 EVENTS
 reaching;
 SWITCHES
 Move, Uturn: reaching;
 Uturn, move;
 COMMUTATION
 move.x = x;
 move.v = v;
 x = move.x;
 Uturn.v = v;
 v = Uturn.v;
 reaching.x = x;
 reaching.v = v;
 reaching.m0x = man0Phase.x;
 reaching.m0v = man0Phase.v;
 reaching.m1x = man1Phase.x;
 reaching.m1v = man1Phase.v;
 END;
```

Note that the design of the “Fly" component is more complex than the design of the “Man", so the corresponding descriptor turned out to be more complicated. Let's pay attention to the SWITCHES paragraph – this is the equivalent of the Table of Switches in the database and the typical characterizations $\left\{sw_j \subset E_j \times M_j \times M_j\right\}_{j=1}^{N}$ with the R_8 axioms in the species of structure description

You will notice that commutation operators are starting to play an increasing role in component descriptors. In order not to multiply the number of such operators, one can enlarge the type descriptions and try to commute large aggregates of variables. In describing the species of structure, there are correlations between the typical characterizations $\left\{m_{j,in} \subset M_j \times \beta(X)\right\}_{j=1}^{N}$, $\left\{m_{j,out} \subset M_j \times \beta(X)\right\}_{j=1}^{N}$, $\left\{e_{j,in} \subset E_j \times \beta(X)\right\}_{j=1}^{N}$, and the axioms $R_5 - R_7$. In the database, commutation operators correspond to the tables of input and output commutations.

Next, we give tables of input and output commutations, but for the whole complex of the von Neumann’s fly. Therefore, first, we give the LCCD descriptor of this complex.

```
COMPLEX menANDfly;
COMPONENTS
 Fly(1), Man(2);
 COMMUTATION
 Fly(0) .man0Phase.x=Man (0).x;
 Fly(0) .man0Phase.v=Man (0).v;
 Fly(0) .man1Phase.x=Man (1).x;
```

```
Fly(0) .man1Phase.v=Man (1).v;
END;
```

The external variables of the components here, on the contrary, can quite legally be on the left side of the commutation operators. On the right side, there are internal variables of the components that calculate the variables of the left side.

TABLE OF INPUT COMMUTATIONS

No	Method	Input offset	Phase offset	Length
1	1	0	16	8
2	1	8	24	8
3	2	0	32	8
4	2	8	40	8
5	3	0	0	8
6	3	8	8	8
7	4	0	8	8
8	5	0	0	8
9	5	8	8	8
10	5	16	16	8
11	5	24	24	8
12	5	32	32	8
13	5	40	40	8

The first column contains the keys. The second column contains the secondary keys that indicate the position of the item in the Table of Methods and Events. The third column contains the offset in bytes from the start of the parameters record of the method. The fourth column contains the offset in bytes from the start of the characteristics record. The fifth contains the length of the parameter in bytes. If we replace the number 8 by 48, at the intersection of the 8th row with the last column – rows 9-13 can be removed. This speaks of the scope for optimization of the LCCD compiler.

TABLE OF OUTPUT COMMUTATIONS

No	Method	Output offset	Phase offset	Length
1	1	0	16	8
2	2	0	32	8
3	3	0	0	8
4	4	0	8	8

The Table of Output Commutation is just as the Input one, only smaller. The first column contains the keys. The second column contains the secondary keys that indicate the position of the item in the Table of Methods and Events. The third column contains the offset in bytes from the start of the parameters record of the method. The fourth column contains the offset in bytes from the start of the characteristics record. The fifth contains the length of the parameter in bytes.

We see that the ontologies of a wide class of modeling domains are representable by combining a database of a simple structure (almost all of its tables were illustrated by the example of von Neumann's fly model) with a not too complex universal program for the computation organization, which provides all the dynamics. The algorithm of the universal program follows from the closeness hypothesis; it is four-cycle, as shown in Fig. 1. This algorithm was described in detail in [2, 4], here we will not repeat this description.

C. DATABASES AND PROGRAMMING TRICKS

Of course, the system dynamics is very important, but if the program for the simulation computations organizing is written, debugged, works, and never changes, – we can forget about it. Then it turns out, that the database forms the base of the conceptual model of almost any domain, and an important question becomes the optimization of the database work and the convenience of dealing with it – the quality of the DBMS.

Here we will present some of the programming solutions implemented in the creation of the MISS (multilingual instrumental simulation system) over 30 years ago [7]. They helped to solve many still urgent problems, including databases and DBMS [8], however, the use of such methods in subsequent years is still not known to us.

The development of the MISS system was carried out on the PC XT. One of the serious problems of this architecture is the impossibility of direct use of computer memory over 640 K, even if such memory presents. Operating systems running in protected mode appeared in the 90s, after the end of the development of MISS.

The way out of this difficulty was to create a software system of virtual memory. The available RAM over 640 K and the disk memory were divided into pages of 64 K. A part of available for addressing 640 K, served as a virtual pages display window. There could be up to 256 virtual pages in each of the 8 memory classes. Thus, it was possible to work with no more than 128 M of virtual memory. Now, this is of course quite a bit, – we consider 2 G of RAM as the minimum for a laptop. However, in the 80s 100 M was considered a very

good removable disk capacity, even for the mainframes. Much more important is how it was possible to work with this virtual memory.

Each byte of the virtual memory could have a constant 32-bit address. Two bytes – for the address on the virtual page, another byte – the page number, 3 more bits - the memory class number, the rest – for the system needs. The constancy of the address means that it continues to work the next time the program is started (if the previous one is completed normally). The virtual address has always been the address of a particular data type defined earlier. This made it easy to find the type of addressable record in the database and use the appropriate methods to work with it.

A library of modules (a set of classes, in modern talking) was implemented for working with virtual memory by virtual addresses - allocating and freeing memory, copying to and from a record, etc. The library included various programming tools: work with lists; with blocked lists (a way to speed up the work of lists); storage in virtual memory, loading, and execution of computer programs (recall that methods and events are included in the base sets of the model-component); working with pictures – means of forming video frames in virtual memory.

What could this give for the creation, management, and operation of databases? For example, in relational databases, a lot of time is spent searching by secondary keys. In the examples in the previous subsection, many tables consist almost entirely of secondary keys. In the MISS database, all secondary keys were virtual addresses of the corresponding tuples, which were accessed simply by the virtual address - without any lookup. Associating an address with the type of what it addresses allowed the creation of generic DBMSs. For example, in MISS, the model database was created automatically as a result of descriptors compilation and model assembly operation. As a result, the database editor had access not only to the fields of tuples of built-in types (integer, double, Boolean, etc.) but also to much more complex fields. For example, if the field type was the virtual address of a list, it was possible to travel through the records of this list, further expanding any of the fields of these records. Or, if the field is a picture, it was opened in a graphical editor. The database of international flights in the former USSR, implemented using MISS [8], worked rather quickly on a not very powerful laptop based on 80386.

Any complex structure based on the virtual addressing may be saved to disk instantly. To do this, you need to rewrite all virtual pages from RAM to disk and remember the states of the display window and the processor. To restore it from the disk instantly – you need to write the necessary virtual pages into the display window and restore the processor state. This solves the problem of data serialization / deserialization (why does my computer boot / shut down so slowly?!), on which the developers of operating systems have been struggling for many years.

D. Dreams of Integration with the Hardware

Remembering how fruitful was the combination of the PDP-11 architecture with the capabilities of the C language, we can dream of including some Model-Oriented Programming ideas into hardware.

The first candidate is a computation organization program that provides all the system dynamics.

Since memory virtualization is somehow inherent in almost all modern processors – the organization of virtual addressing available to the developer is rather a software question.

Finally, why not dream of the operating system kernel as a model-component based on a hardware computational organization. After all, the functions they perform are the same – the organization of the reactive behavior for the system. Of course, there are a lot of unresolved issues here, such as those related to security, and some others. However, there are appropriate specialists for this, who could be involved in the project.

IV. Conclusion

Model-Oriented Programming is radically different from the well-known concepts of software systems model-driven development – MDA (Model-Driven Architecture), MDE (Model-Driven Engineering), and MDD (Model-Driven Development), despite the similarity of the names. The latter are just add-ons to the Object-Oriented approach and do not take the programming out from the field of art to the field of technology. The main development tool in these

concepts is still imperative programming in languages like the C family or Java.

Model-Oriented Programming fully implements the CAD ideas in the programming, with all their advantages and disadvantages. At the same time, programming at all levels becomes fully declarative, which greatly facilitates the subsequent debugging of the software system.

The software system is seen as consisting of "atoms" – models-components, which can be combined into complexes, which themselves, being model-components, can be combined into complexes of a higher level, etc. Comparing with the Object-Oriented approach, we can say that in the Model-Oriented Programming the multiple inheritances from the bottom to the top dominate, i.e. the Model Synthesis.

A declarative programming language LCCD (language for components and complexes describing) has been developed (an analog of N. Bourbaki's species of structure, which is closer to the perception of the programmer). It describes the structure of the model-components and the formation of complexes from them. An important feature of Model-Oriented Programming is that the LCCD descriptors are compiled not into computer code, but into a database. The UML and related concepts of model-driven software development have been killed by the poor code quality after double compilation. In MO programming, the question of code quality is not an issue - language constructs are compiled into the database either right or wrong. The efficiency of computations depends on a universal program for organizing computations, which can be "polished to shine". The database is the third (with the LCCD) analog of the universal agent description by N. Bourbaki's species of structure.

It should be noted, that the end-to-end technology described above for describing, synthesizing, and software implementation of simulation models of complex multicomponent systems is not just a theoretical dream. On the contrary, it all started with the complete practical implementation of such a system [7] (with the declarative programming language of the LCCD type, compilation, debugging, database, and, of course, a program for the calculations organizing) in the MS-DOS environment.

The successful database design based on the virtual addressing technology, using this system was described in [8].

This system took first place in the category of professional programs in the All-Union competition of computer programs held in the USSR by the Japanese company "ASCII corporation" (at that time the largest computer games developer in Japan, now a division of a publishing company), in 1990. True, there was no at all any theoretical substantiation of this technology then, – it was given later in [4] – only the practical realization.

REFERENCES:

[1] C. Böhm, G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Communications of the ACM*, Vol. 9, No 5, 1966, pp. 366–371. DOI:10.1145/355592.365646.

[2] Yu. I. Brodsky, Elements of Geometric Theory of Complex Systems Behavior // *WSEAS Transactions on Systems and Control*, Vol. 15, 2020, pp. 19-29. DOI:10.37394/23203.2020.15.3.

[3] N. Bourbaki, *Elements of Mathematics. Theory of Sets*, Springer, 2004. 414 p. DOI:10.1007/978-3-642-59309-3.

[4] Yu. I. Brodsky, Bourbaki's Structure Theory in the Problem of Complex Systems Simulation Models Synthesis and Model-Oriented Programming, *Computational Mathematics and Mathematical Physics*, Vol. 55, No 1, 2015, pp. 148-159. DOI:10.1134/S0965542515010054.

[5] Yu. I. Brodsky, Model Synthesis and Model-Oriented Programming a new technology for high performance agent-based modeling // *Proceedings of the 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC), Vladivostok, Russia.* IEEE: 2018. 8482121. DOI:10.1109/RPC.2018.8482121.

[6] N. P. Buslenko, Complex systems and simulation models, *Cybernetics*, Vol. 12, No 6, 1976, pp. 862-870. DOI:10.1007/BF01070419.

[7] Yu. I. Brodsky, V. Yu. Lebedev, *Instrumental'naya sistema imitatsionnogo modelirovaniya MISS* [Instrumental Simulation System MISS] Moscow: CC AS of the USSR, 1991, 180 p. (in Russian) http://www.ras.ru/ph/0005/VDJBWF5N.pdf

[8] Yu. I. Brodsky, V. Yu. Lebedev, O technologii razrabotki baz dannych na osnove instrumental'noj sistemy MISS [On the technology of database development based on the MISS instrumental system] // *Modelirovanie, dekompozitsija i optimizatsija slozhnych dinamicheskich protsessov* [Modeling, decomposition, and optimization of complex dynamic processes], Vol. 11, No 1-1(11), 1996, pp. 61-67. (in Russian) http://www.ras.ru/ph/0005/P60GG76A.pdf