The theory graph modeling systems from quality modules of the application areas[1]

E. M. Lavrischeva
Doctor of phys.-мath. Sci., Professor of MIPT,
scientifically specialist ISPRAS.
Lavryscheva@gmail.com, lavr@ispras.ru

Abstract: The graph modeling of applied systems (AS) from ready resources (modules) are presented. The graph is represented by an adjacency and reach ability matrix. A new program structures are modeling by mathematical operations (unions, connections, etc.). The assemble of programs structures (complex, system, packets, AS, OS, IS, etc.) from modules in LP, as resources are integrating by such operations (link, make, weaver, config, etc.) and controlled on the quality every recourses and the making systems from them.

Mathematics is more than science,
It's the language of science.
Niles Bohr

1 Introduce. The graph theory of program

Programming theory is a mathematical science, the object of study of which is the mathematical abstraction of the functions of programs with a certain logical and information structure, focused on computer execution. With the advent of the LP began to develop new methods of analysis of algorithms of problems of AS, decomposition of areas into separate functional objects, displaying them in the vertices of the graph to create a complex structure of AS (complex programs, aggregate, large program, system, etc.). Functional elements of missile defense were first called modules, programs, then objects, components, services, etc.

A module is a formally described program element that displays certain AS function that has the property of completeness and connectivity with other elements according to the data specified in the interface part of the description. From a mathematical point of view, a module is a mapping of a set of initial data X to a set of output Y in the form $M: X \rightarrow Y$.

A number of restrictions and conditions are imposed on X, Y and M to make the modules an independent program element among other types of program objects [1-3].

Types of connections between modules via input and output parameters are as follows:

1) linking of control: $CP = K_1 + K_2$, where K_1 is the coefficient of the calling mechanism; K_2 is the coefficient of transition from the environment of the calling module to the environment of the called;
2) Linking of data: $CI = \sum_{i=1}^{n} K_i F(x_i)$, where *Kid* - the weight coefficient *i*ron of the parameter; $F(x_i)$ – the element function for the parameter x_ii.

Coefficients $K_{id} = 1$ – for simple variables and $K_{id} > 1$ – for complex variables (array, record, etc.). $F(x_{.i}) = 1$ if x_i - a simple variable and $F(x_i) > 1$ if complex.

The program, modular structure is given by the graph $G = (X, \Gamma)$, where X - a finite set of vertices; Γ - a finite subset of the direct product of X z on the set of relations on the arcs of the graph. The program structure represents a pair $S = (T, \chi)$, where T - a model of a program, modular structure; χ - a characteristic function given on the set of vertices X of the graph G.

The value of the characteristic function χ is defined as:

- X(x) = 1 if the module with vertex $x \in X$ is included in the modular system;
- X(x) = 0 if the module with vertex $x \in X$ is not included in the modular system and is not accessed from him.

Definition 1. Two models of program structures $T_1 = (G_1, Y_1, F_1)$ and $T_2 = (G_2, Y_2, F_2)$ are identical if $G_1 = G_2$, $Y_1 = Y_2$, $F_1 = F_2$. The T_1 model is isomorphic to the T_2 model if $G_1 = G_2$ between sets Y_1 and Y_2 exists an isomorphism φ, and for any $x \in X$ $F_2(x)=\varphi(f_1(x))$.

[1] Development on Project RFFI № 19-01-00206

Definition 2. Two program structure $S_1 = (T_1, \chi_1)$ and $S_2 = (T_2, \chi_2)$ are identical if $T_1 = T_2$, $\chi_1 = \chi_2$ and the structures S_1 and S_2 are isomorphic, then T_1 is isomorphic to T_2 and $\chi_1 = \chi_2$. The concept of isomorphism of program structures

and their models is used in the specification of the abstraction level at which operations on these structures are defined. For isomorphic graph objects, operations will be interpreted in the same way without orientation to a

specific composition of program elements, provided that such operations are defined over pairs (G, χ). The software module is described in the *LP* and has an interface section in which external and internal parameters are set for data exchange between related modules through Call/RMI operations, etc.

The interface defines the connection of heterogeneous software modules according to the data and the way they are displayed by programming systems with the *LP*. Its main functions are: data transfer between program elements (modules), data conversion to the equivalent form and transition from the environment and platform of the called module to the caller and back. Functions of conversion of different, non-equivalent data types is carried out with the help of a previously developed library of 64 primitive functions for heterogeneous types of data of *LP* in the APROP system [1-5] and included in the common system environments of the OS (IBM, MS, Oberon, UNIX, etc.).

In practice, the assembly method of software modules is performed by operations (link, make, assembling, config, weaver) special programs OS libraries (OS ES, IBM, MS.Net, etc.), a builder of complex applications in OS RV for SM computers, complication modules for ERM "Elbrus" are used. In these operations and interface functions data type conversion library [2]. Next, we consider the mathematical theory of graphs of software modular structures and mathematical operations (union, projection, difference, etc.) implementation of ways of linking the graph modules and the semantics of the transformation of data transmitted by the vertices of the graph.

1.1 Definition of a modular structure graph

To represent modular structures, we use the mathematical apparatus of graph theory, in which the graph G is treated as a pair of objects G = (X, E), where X - a finite set of vertices, and E is a finite subset of the direct product of X × X × Z - arcs of the graph, corresponding to a finite vertex (Fig. 1).

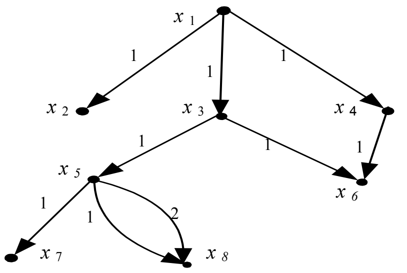

Figure 1. Graph of program from modules with arcs 1, 2

The set of arcs of the graph have the form: $E = \{(x_1, x_2, 1), (x_1, x_3, 1), (x_5, x_8, 1), (x_5, x_8, 2)\}$. Based on this definition, we can say that the graph G is a multi-graph, since its two vertices can be connected by several arcs.

To distinguish these arcs introduced their numbering positive integers – 1, 2. (Fig.1) and vertices of the graph $x_1, x_2, ..., x_8$ form a set of *X*.

From the module corresponding to the vertex x_5, there are two calling operators to the modules, with vertices x_7, x_8.

Definition 3. A program aggregate is a pair S = (T, χ), where E – a model of the program modular structure of the aggregate; χ - a characteristic function defined on the set of vertices X of the graph of the modular structure G. The value of the χ function is defined as follows:

χ(x) = 1 if the module corresponding to the vertex x ∈ X, - included in the unit;

χ(x) = 0 if the module corresponding to the vertex x ∈ X, - not included in the software unit, but it is accessed from other modules previously included.

Definition 4. The model of the program structure of the program unit is an object described by the triple T = (G, Y, F), where G = (X, E) - a directed graph, which is a graph of a modular structure;

Y is a set of modules included in the program aggregate;

F is a correspondence function that puts an element of the set y at each vertex X of the graph.

$$\text{Function F maps X to Y,} \quad F : X \rightarrow Y \qquad (1)$$

In General, an element from Y can correspond to several vertices from the set X (which is typical for the dynamic structure of the aggregate) [3].

The graph of software aggregates has the following properties:

1) graph G has one or more connectivity elements, each of which represents an acyclic graph, i.e. does not contain oriented cycles;

2) in each graph G is allocated a single vertex, which is called the root and is characterized by the fact that there are no arcs included in it and the corresponding module of the software unit is performed first;

3) cycles are allowed only for the case when some vertex has a recursive reference to itself. Typically, this feature is implemented by the compiler with the corresponding LP and this type of communication is not considered by the intermodule interface. Therefore, such arcs are not included in the graph. The exception to the consideration of other types of cycles is due to the fact that some modules will have to remember the history of their calls in order to return control correctly, which contradicts the properties of the modules;

4) An empty graph G_0 corresponds to an empty program structure.

Next, the graph G will be used to illustrate mathematical operations on modular structures. On Fig.2. four types of subgraphs are shown for Fig.1. For the graph (Fig.1) set of subgraphs (from left to right) to create four programs: 1) from the vertex x_1 given three related vertices x_1, x_3 and x_4; 2) for x_3 given two vertices x_5 and x_6; 3) for x_5 given two vertices x_7 and x_8; 4) to the program x_5 added vertex x_6, not associated with it.

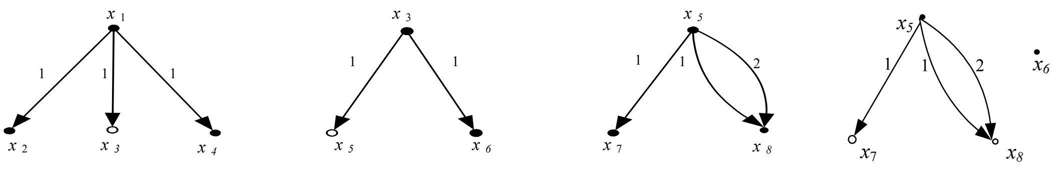

Figure. 2. The graphs of modules structures

A subgraph - a fragment of a software aggregate $G^s = (X^s, E^s)$ for whose functions one of two conditions is satisfied:

$C(S) = 1$, if $\chi(x) = 1$ for any x of X;

$C(S) = 0$, if there is x such that $\chi(x) = 0$;

$R(S^s) = 0$, if the modular structure is part of a higher-level structure and $R(S)=1$ if the software assembly is ready to run.

Given these combinations C and R, the subgraph can be: open ($C = 0, R = 0$); closed at the top ($C = 0, R = 1$); closed at the bottom ($C = 1, R = 0$).

1.2 The graph of the modules for Software structure

The graph of the module (m) is represented as: $G^m = (X^T, E^T)$. It contains a single vertex $x \in X^T$ for which $\chi(x_j)=1$. This vertex is the root. An arc of the form (x_j, x_e, k) means calling the module to the corresponding vertex x_j, i.e. to the module with the vertex x_l. The dark circle on the graph corresponds to the vertex for which $\chi(x) = 1$;

light – $\chi(x)=0$.

Program graph $G^p = (X^p, E^p)$ which is performed $C(S^p) = 1$; $R(S^p) = 1$. An example of a graph of such a program modular structure is shown in Fig. 1.

The graph of the complex $G^c = (X^c, E^c)$ consists of n connectivity components ($n > 1$), each of which is a graph and includes: $G^c = G_1^p \cup G_2^p \cup, \ldots, \cup G_n^p$,

where $X^c = X_1^p \cup X_2^p \cup, \ldots, \cup X_n^p$ и $E^c = E_1^p \cup E_2^p \cup, \ldots, \cup E_n^p$.

These definitions of the graph of the program module, program and complex are used for the process of assembling the modules. These concepts may differ from similar ones, which are considered in other contexts of the work.

1.3 Matrix representation of graphs from program elements of module types

To determine the main operations on software structures, we use the mathematical apparatus of the matrix representation of graphs in the form of an adjacency and reachability matrix. That is, the graph is represented by the matrix $M = m(i, j)$ of adjacency and is proved by the reachability matrix [1-8]. The element of the matrix m_{ij} determines the number of call operators with index i, to the module with index j.

In addition to the adjacency matrix (calls), the characteristic vector $V_i = \chi(x_i)$ for i-elements is used. For a modular structure graph (Fig. 1) characteristic vector and adjacency matrix have the form:

$$V = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \qquad M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2}$$

We analyze adjacency matrices and characteristic vectors for subgraphs and graphs of modular structures corresponding to different types – program, complex, aggregate, etc. For subgraphs (Fig.2) vectors and matrices have the form:

$$V_3^s = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad M_3^s = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; \qquad V_1^s = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$M_1^s = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}; \quad V_5^s = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad M_5^s = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{3}$$

For the program graph (Fig. 1) the characteristic vector and the matrix of calls coincide with V and M, respectively, and determine the form (2), in which all elements of V are equal to one. In the case of the complex, the characteristic vector and the call matrix have the following form:

$$V^c = \begin{pmatrix} V_1^p \\ V_2^p \\ \dots \\ V_n^p \end{pmatrix}, \quad M^c = \begin{pmatrix} M_1^p & 0 & \dots & 0 \\ 0 & M_2^p & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & M_n^p \end{pmatrix} \tag{4}$$

Here V_i^p and M_i^p $(i = \overline{1, n})$ denote the characteristic vector and the adjacency matrix for the graph of the i-th program included in the graph of the complex. In the future, the matrix representation is used when performing mathematical operations on software structures.

1.4 The relation of the reach ability graph of program structures

Let $G = (X, E)$ - a graph of a program of modular structure; x_i, x_j - vertices belonging to X. If there is an oriented chain from x_i to x_j in the graph G, then the vertex x_j is reachable from the vertex x_i. The following statement is true: if the vertex x_j is reachable from x_l – из x_j, x_l – from x_j , then x_l is reachable from x_l. The proof of this fact is obvious.

Consider a binary relation on the set X that determines the reach ability of one vertex of a graph to another. We introduce the notation $x_i \rightarrow x_j$ - reach ability of the vertex x_j from x_i. The relation is transitive. Denote by $D(x_i)$) the set of vertices of graph G reachable from x_i.. Then the equality

$$\overline{x_i} = \{x_i\} \cup D(x_i) \tag{5}$$

of determines the transitive closure of x_i in relation to the achievability of tops. We prove the following theorems.

Theorem 1. For the selected element of connectivity of the graph of the program structure, any vertex is reachable from the root corresponding to the given vertex of the graph, i.e. the equality (x_1 – root vertex).

$$\overline{x_1} = \{x_1\} \cup D(x_1) = X. \tag{6}$$

Evidence. Suppose the vertex x_i $(x_i \in X)$ is unattainable from x_1. Then $x_i \notin \overline{x_1}$ and the set X' = $X \setminus \overline{x_1}$ - not empty. Since the selected component of the graph is connected, there is a vertex $x_j \in \overline{x_1}$ and a chain $H(x_i, x_j)$, leading from x_i to x_j. Based on the acyclicity of the graph G, in X'' there should be a simple chain $H(x_l, x_j)$, where the vertex x_l does not include arcs (this chain can be empty if X' consists only of x_i). Consider the chain $H(x_l, x_j) = H(x_l, x_i) \cup H(x_i, x_j)$. This means that the module x_i is reachable from vertices x_1 and x_i and both vertices contain no incoming arcs. This contradicts the definition of a graph of a modular structure with a single root vertex. The theorem is proved.

The results of this theorem are important to substantiate the requirement of the absence of oriented cycles in the graph of the program structure with respect to the notion of reachability. Consider the graph shown in Fig. 3.

From this figure it is clear that the graph contains a directed cycle and modules corresponding to vertices x_4, x_5, x_6 will never be executed.

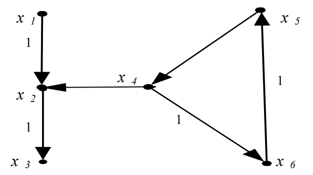

Figure. 3. A graph contains directed cycle

Thus, the results of theorem1 reinforce the requirement that there are no oriented cycles in the graph of the program.

We analyze the matrix representation of the reach ability relation for the graph of the program structure Fig.1 with the reach ability matrix A, which has the form (7).

Coefficient a_{ij} = 1 if the module corresponding to the index 1 is reachable from the module corresponding to the index i the following results are based on the theorem from graph theory.

$$A = \left(\begin{array}{cccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{7}$$

Theorem 2. The coefficient m_{ij} of the l-th degree of the adjacency matrix M^l determines the number of different routes containing l arcs and connecting vertex x_i to the vertex of the x_j –oriented graph. The proof of this theorem is given in [6]. Consider the following three consequences of this theorem.

Corollary 1.1. Matrix $\overline{M} = \sum_{l=1}^{n} M^i$, where M is the adjacency matrix of a directed graph with n vertices coincides up to the numerical values of the coefficients with the reachability matrix A.

Evidence. In a directed graph containing n vertices, the maximum path length without repeating arcs cannot exceed n. Therefore, the sequence of degrees of the adjacency matrix M^i, where i = 1,2, ..., n determines the number of all possible paths in the graph with the number of arcs ≤ p. Let the coefficient $\overline{m_{ij}}$ of the matrix M be different from zero. This means that there is a degree of matrix M^i in which the corresponding coefficient $\overline{m_{ij}}$ is also nonzero. Therefore, there is a path from vertex x_i to x_j, i.e. vertex x_j is reachable from x_i. This consequence determines the connection of the matrix of calls of the graph of the modular structure M, coinciding with the reachability matrix A, and determines the algorithm for constructing the latter.

Corollary 1.2. Let there be a coefficient $m_{ii} > 0$ for some i in the sequence of degrees of the adjacency matrix M^i. Then there is a cycle in the original graph.

Evidence. Let $m_{ii} > 0$ for some l. Therefore x_l reachable from x_i, i.e. there is a cycle. According to the theorem, this cycle has l arcs (generally repeated).

Corollary 1.3. Let the n-th degree of the adjacency matrix of the M^n of the acyclic graph coincide with the zero matrix (all coefficients are zero).

Evidence. If the graph is acyclic, then the simplest path cannot have more than $n - 1$ arcs.

If M^n has a coefficient other than zero, then there must be a path consisting of n arcs. And this way can only be oriented cycle. Therefore, all coefficients of M^n for an acyclic graph are zero. This consequence provides a necessary and sufficient condition for the absence of cycles in the graph of a modular structure.

For acyclic graphs, the reachability ratio is equivalent to a partially strict order. The transitivity of the reachability ratio was considered above. Anti-symmetry follows from the absence of oriented cycles: if the vertex x_j is reachable from x_j, then the opposite is not true.

We introduce the notation $x_i > x_j$ if vertex x_j is reachable from vertex x_i.

Let G = (X, E) be an acyclic graph corresponding to some program structure.

Consider the decreasing chain of elements of a partially ordered set X: $x_{i1} > x_{i2} > ... > x_{in}$. …,

where " > ” denotes the reachability ratio.

Since X is finite, the chain breaks. The vertex x_{in} has no outgoing arcs, i.e. the element x_{in} is minimal (it corresponds to a module that does not contain access to other modules). The maximum element in the set X is the root vertex.

2 Mathematical operations on the graph elements

Mathematical operations (U, ∩, /, +, -) on graphs are performed at the level of abstractions of elements of program structures that lead to changes in graph elements and characteristic functions of systems: S = (G, χ).

Let $S_1 = (G_1, \chi_1)$ and $S_2 = (G_2, \chi_2)$ be two graphs of program structures $G_1 = (X_1, E_1)$ and $G_2 = (X_2, E_2)$ respectively.

We introduce the following notations:

$D(x)$ – the set of vertices reachable from the vertex x;

$D^*(x)$ – the set of vertices from which vertex x is reachable.

The same symbols are used for the same vertices included in the graphs G_1 and G_2. The main operations on the program structures are discussed below

Merge (join) operation $S = S_1$ U (9)

is intended to form a graph of the structure of the complex and is formally defined as follows S_1 and S_2 – any program structures that satisfy the definitions of claim 1:

$$G = G_1 \oplus G_2, \quad X = X_1 \oplus X_2, \quad E_1 \oplus E_2, \tag{10}$$

where the symbol denotes a direct sum provided:

$$\chi(x) = \chi_1(x), \text{ if } \chi \in X_1,$$
$$\chi(x) = \chi_2(x), \text{ if } \chi \in X_2.$$

The same vertices included in G_1 and G_2 are represented by different objects in the operations of combining program structures. The characteristic vector and adjacency matrix of the program structure S are defined as follows:

$$V_{1,2} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, M_{1,2} = \tag{11}$$

where $V_{1,2}$ and $M_{1,2}$ are characteristic vectors and adjacency matrices of modular structures S_1 and S_2 respectively. This operation is associative, but not commutative – the order of the operands determines the order of the components of the complex.

It should be noted that if the operands S_1 and S_2 satisfy the conditions for defining program structures, the result S will also satisfy the same requirements. The join operation increases the number of connected graph elements. In addition, the column structures may themselves have multiple items of connectedness. For the rest of the operation counts of the operands and result are the only element of connection.

The connection operation. We denote by x_i and x_j the root vertices of graphs G_1 and G_2 of program structures S_1 and S_2, respectively. This operation

$$S = S_1 + S_2 \tag{12}$$

which is execute if these structures meet the following conditions:

$$\begin{gathered} \text{- set } X' = X_2 \cap X_2 \text{ not empty;} \\ \text{- vertex } x_j \in X' \text{ and } \chi(x_j) = 0; \\ \text{- } D^*(x) \cap D(x) = 0 \text{ for every } x \in X', \text{ where } D^*(x) \in X_1 \text{ и } D(x) \in X_2; \\ \text{- } G = G_1 \cup G_2, \quad X = X_1 \cup X_2, \quad E = E_1 \cup E_2, \end{gathered} \tag{13}$$

The characteristic function χ is satisfied under the condition:

$$\begin{gathered} \chi(x) = \chi_1(x), \text{ if } x \in X_1 \setminus X'; \\ X(x) = max(\chi_1(x), \chi_2(x))> \text{ if } x \in X', \\ \chi(x) = \chi_2(x), \text{ if } x \in X_2 \setminus X'. \end{gathered}$$

First condition means that there are common vertices in graphs G_1 and G_2. According to the second condition, the root vertex G_2 belongs to the common part and for S_1 the object corresponding to x_j is not included in the program structure yet.

The third condition prohibits the existence of cycles in the result graph. Indeed, if there is $x_n \in D^*(x) \cap D(x)$, then $x_n > x$ and $x > x_n$, and $x > x_n$, then this means the existence of a cycle.

If S_1 and S_2 satisfy the above conditions, the connection operation is partial.

Let us determine whether the result of the connection operation belongs to the class of program structures. Since X'' is not empty, the graph G has one connected component. The root vertex of the graph G is x_i. The graph G itself has no oriented cycles, i.e. is acyclic.

Thus, S belongs to the class of program structures under consideration.

This connection operation is not commutative and is generally not associative. To show that this operation is not associative, consider the result $S = (S_1 + S_2) + S_3$, where the root vertices of graphs G_2 and G_3 are part of the vertices of graph G_1 and $X_2 \cap X_3 \neq 0$.

Then the result of the $S_2 + S_3$ join operation is undefined.

The operation of projection. Let $S_1 = (G_1, \chi_1)$ be a program structure and $x_i \in X_1$. The operation of projection of this structure to the top of the graph S_1 is denoted as $S = P_{rxi}(S_1)$ and is defined as

$$G(X, \Gamma), \quad X = \overline{x}_i, \quad E = \{(x_i, x_j, K) | x_i, x_j \in X\}, \tag{14}$$

and for the characteristic function is $\chi(x) = \chi_1(x)$, if $x \in X$. The projection operation defines the program structure S_1 in the structure S. let's check the belonging of the structure S to the class of the considered program structures. If the graph of the structure S_1 is connected acyclically, then the same properties will be possessed by the graph G. There is a single root vertex x_i in the graph G. Thus, the program structure S belongs to the class under consideration.

The difference operation for program structures is defined as follows. Let $S_1 = (G_1, \chi_1)$ be a program structure and $x_i \in X_1$. The difference operation is performed on this structure and its projection to the vertex x_i of the corresponding graph (x_i is not the cortical vertex of the graph G_1). Formally, the difference operation of the program structure has the form:

$$S = S_1 - P_{r_{xi}}(S_1 \tag{15}$$

and defined as follows

$$\begin{gathered} G = \{X, E), \quad X = (X_1 \setminus \overline{x_i}) \cup X' \\ E = \{(x_i, x_j, K) | x_i, x_j \in X\}, \end{gathered} \tag{16}$$

where the set X' consists of such elements for which

$$X' = \{x'_j | (x_l \in X_1 \setminus x_i) \;\&\; (x'_j \in \overline{x_i}) \;\&\; (x_l, x'_j, K) \in E \tag{17}$$

Here, the characteristic function χ is defined as:

$$\begin{gathered} \chi(x) = \chi_1(x), \text{ если } x \in X_1 \setminus \overline{x_i}; \\ \chi(x) = 0), \text{ если } x \in X'. \end{gathered}$$

The set X includes vertices that are not included in the set $\overline{x_i}$, and those vertices $\overline{x_i}$ that include arcs from vertex $X_1 \setminus \overline{x_i}$ (sets X'). The characteristic function for elements $x' \in X'$ is zero. The difference operation is the inverse of the join operation, i.e. the equality is performed:

$$S - P_{r_{xi}}(S) + P_{r_{xi}}(S) = S. \tag{18}$$

Let us check that S, defined in (15), belongs to the class of program structures. If the graph is G, connected and acyclic, then the graph G_1 will have the same properties. The root vertex G is the same as the root vertex G_1. Thus, S satisfies the conditions for determining the program structure given in paragraph 1.

Let S* be the set of program structures given by the direct product G* X χ*, where G^* and χ^* are the set of graphs and the set of characteristic functions. Denote by $\Omega = \{U, \cap, /, +, -\}$ - set of mathematical operations on program structures and *P, C* and *R* - predicates of:

$$\Omega = \{U, \cap, /, +, - , P, C, R \qquad (19)$$

Thus, an algebraic system $\Sigma = (S, \Omega)$ over a set of program structures and operations on them (union, connection, differences and projections) is defined.

2.1 Characteristics of simple and complex graph structures

Among the variety of program structures there are three main ones – a simple, complex structure with a call of modules from the external environment and a dynamic structure. The main purpose of various structures is the most optimal use of the main memory during the execution of the unit.

Simple structure. An aggregate with a simple structure is created in the process of building modules based on the operations of link calls. The amount of main memory occupied by an aggregate with a simple structure is constant and equal to the sum of the volumes of individual modules: $V_s = \sum_{i=1}^{n} v_i$, where v_i is the amount of memory occupied by the *i*-th module. The corresponding graph of a modular structure is always connected.

Complex structure. Assembly of complex structures with dynamic invocation of modules in the shared memory is created in the Assembly process of the modules. In such an aggregate, the connections between the modules are not so rigid and their sequence is determined by the modules included in the chain. The modules are loaded into the main memory at the time of processing. When finished, the memory is freed and used to load another module. The graph of a complex program structure is also connected (Fig.4) and is reflected in the matrix (2).

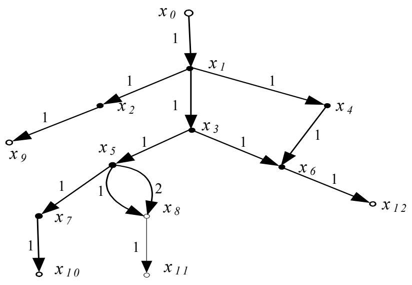

Figure 4. Modification graph of program structure

The amount of main memory required depends on the number and composition of modules and the maximum amount of memory is equal to the sum of individual modules:

$$v_0^{\max} = V_s = \sum_{i=1}^{n} v_i .$$

The minimum amount of memory required when performing the aggregate is calculated by Floyd's algorithm, which determines the shortest path in the graph, in which each arc corresponds to a weight coefficient, called the arc length. The following transformations are performed to apply the Floyd algorithm.

1). Let's add new vertices and arcs to the graph. The vertices are $x_0, x_{n+1}, \dots , x_{n+m}$,

where *m* is the number of end vertices. New arcs include $(x_0, x_1, 1)$, $(x_{r1}, x_{n+1}, 1)$, ..., $(x_{rn}, x_{n+rn}, 1)$. In them x_1 corresponds to the main module and all x_i – to the end vertices. After performing operations, the graph of the modular structure (Fig. 1) is given to the graph on Fig. 5 with vertices $x_0, x_9, x_{10}, x_{11}, x_{12}$. It vertices correspond to the weight coefficients:

$$v_0 = v_9 = v_{10} = v_{11} = v_{12} = 0$$

2). Each arc of the form (x_i, x_j, k) is assigned a coefficient $v_{ij} = \frac{v_i + v_j}{2}$.

Consider all routes leading from x_0 to one of the other additional vertices. The length of the shortest route path is calculated as follows:

$$l_{0,n+p} = v_{01} + \dots + v_{rp,n+p} = \frac{v_0 + v_1}{2} + \dots + \frac{v_{2p} + v_{n+p}}{2} = \frac{v_0}{2} + v_1 + \dots + v_{rp} + \frac{v_{n+p}}{2} = v_{1+} \dots + v_{rp}.$$

This length $l_{0,\,n+p}$ will be equal to the sum of the memory modules for path $x_1, \dots, x_{rp}$.

Thus, applying Floyd's algorithm to the graph in Fig. 2, we solve the problem of calculating the amount of memory for the maximum chain.

3). We replace the adjacency matrix with the path matrix. For each $m_{ij}>0$, the corresponding location will be v_{ij}. The values $m_{ij} = \emptyset$ are replaced by $-\infty$. The program implementing Floyd's algorithm has the following form (it is assumed that the path matrix is described as a two-dimensional matrix $(n \times n)$: this length $l_{0,\,n+p}$ will be equal to the sum of the memory modules for path x_1, x_{rp}.

```
For k = 1 to n do
  For I = 1 to n do
    For j = 1 to n do
      if M[I, j ] < M [ i l k] + M[k, j] then M [I, j]: = M [ i , k] + M[k, j].
```

As a result of this algorithm, a matrix of maximum paths will be constructed. The maximum of $l_{0,\ n+p}$ will determine the minimum amount of $l_{0,\ n+p}$ memory for the memory-overlapping aggregate.

The most complex structure for the values $V_0^{min} \leq V0 \leq V_0^{max}$ can be constructed by following the algorithms proposed in [4-7]. The qualitative dependence of V0 on the number of dynamic sites is shown in Fig.5. Here *n* is the number of modules in the unit. Despite the different kind of curves, they have a common pattern – any V_0 is enclosed between the values of v_0^{max} и v_0^{min}.

Dynamic structure. The mechanism of dynamic links between modules is different from the call mechanism. Dynamic objects are loaded into the main memory when they are accessed. By analogy, we call the volume loaded with a single treatment of a dynamic element, has its own program structure, for which the adjacency matrix is composed. If the same modules are found in different dynamic structures, they are different objects.

The original graph is used for illustration (Fig.1). Let the module corresponding to the vertex x_1, be dynamically called from the module corresponding to the vertex x_3. The resulting modified graph is shown in Fig. 6. A dashed arrow indicates a dynamic call. The module corresponding to the vertex x_6, occurs twice.

We construct an adjacency matrix for this aggregate. Each dynamic element will have its own *CAL.L.* To distinguish a dynamic call, the corresponding matrix elements will contain negative numbers whose absolute values specify the number of dynamic calls between the data of the module pair. The adjacency matrix will look like:

$$M=\begin{pmatrix} x_1 & x_2 & x_4 & x_6 & x_3 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (20)$$

We investigate the qualitative dependence of the amount of RAM on the number of dynamic segments (Fig.5 and Fig. 6).

With one component in the software unit of a simple structure we have $V^1{}_d = V_s$.

If each dynamic component consists of one module, then the modified Floyd algorithm finds the maximum path and $V_d^n = V_0^{min}$.

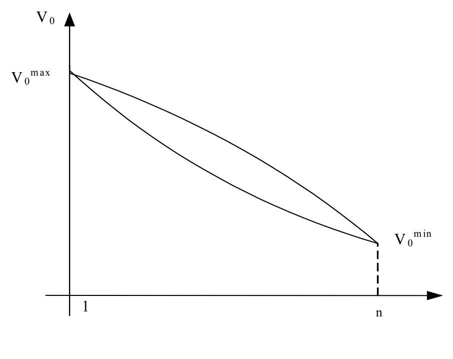

Figure 5. Grafics of qualitative dependence V_a from the number of subgraph

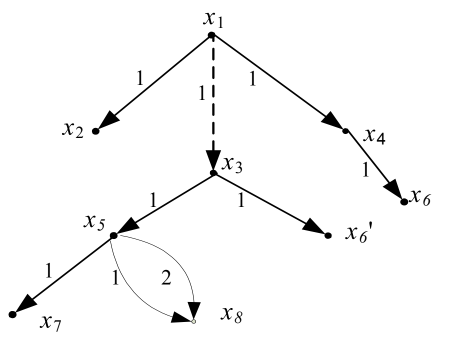

Figure 6. Graph programs structure with dynamic Calls

For intermediate values, the dependence is more complex. On fig.7 presents two curves (1, 2), and n is the number of modules in the program unit.

Due to the duplication of modules there is an increase in the main memory of the OS.

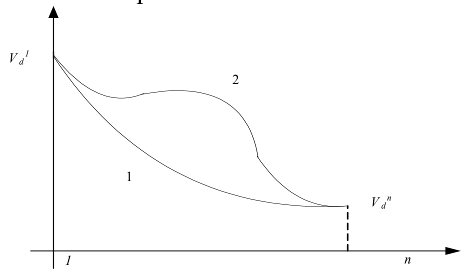

Figure 7. Grafic dependence V_a from the number of dynamic elements

Thus, curve 1 is characteristic of software aggregates of graphs in the form of a tree, which ensures that there are no identical modules in the graph. Despite the lack of dynamic structure in terms of memory savings, there is a significant advantage – independence from editing links. Each dynamic object can be modified, and editing relationships in the OS is not required.

Curve 1 defines a relationship in which different segments do not have the same modules. Curve 2 describes the dependence for the case when different segments have the same modules. For them, the required memory increases due to the duplication of such modules. However, dependence 2 is typical for the case when there are no identical modules in dynamic structures and they are written in high-level LP. These modules are handled by utility tools – memory management, I /o, emergency handling, etc.

3 Operations of assembling elements of graph g

Let the graph G be represented by the set of modules X= $\{x_1, x_2... x_m\}$, as described in PL, and located at the vertices of the graph. The modules are assembled into a software unit. In this case, each pair of modules x_i, x_j *(i, j* – languages from the set of *LP* are connected by the relation of call on the basis of which the module of communication x'_{ij} is formed. In General, for simple program structures, the aggregate contains link communication (call) operators and forward and reverse transformations of data types passed from the calling module (in *i*-language) to the calling module (in *j*-language) and back [23].

LP allows you to describe the information part - passport modules with a description of the transmitted data [8-13] and operations call modules. Taking into account the passports of the modules, the software structure of the unit is built (program - Prog, complex - Comp, package - Pac). The passport describes the special language WSDL containing: - a subset of the operations associate link elements of the graph in the language L' that contains a description of the parameters from the list of actual and formal parameters of the invocation; - mathematical operations on the graph and operations of binding modules in a complex structure (Prig, Comp, Agar, Pack and so on).

The operator modules link (make, config, assembling, etc. since 1994) takes the form:

Link <aggregate type> <aggregate name> (<main module name>, <additional list of module names>) <execution mode>,

when constructing specific program structures, the vertices of the graph – modules can be marked with special symbols ρ, denoting:

* – the beginning of the dynamic fragment with the vertex marked by this symbol;

+ - the module in the graph G is marked as the main program of the complex;

/ - means enabling debugging and so on.

Using these designations, the graph G will take the form shown in figure 8 and has a representation: $E = \{(x_5,x_7,1),\ (x_5,x_8,1),\ (x_5,x_8,2)\}$.

The aggregate is given a unique name corresponding to the generated root module. For the graph $E = \{(x_4, x_6, 1)\}$ a fragment of operators providing a dynamic call will be formed in the communication module x'_{46}. For a pair of modules specified in Fig.8 vertices $x_4, x_6,$ the structure of the corresponding part of the unit, including the communication module, is shown in Fig. 9. Similarly links of modules and other types of calls are implemented.

Thus, for a pair of modules x_i, x_j, a module of connection x_{ij} of the form:

$$x'_{ij} = S_0 * (S_1 \times S_1^T) * (S_2 \times S_2^T) * S_0^1,$$

where S_0 is a fragment of the aggregate that defines the environment of x*j* module functioning;

S_1 – a fragment of the aggregate, including a sequence of calls to functions from the set {P, C, S}, each of which performs the necessary conversion of the actual parameters when referring to the x_j -module;

S_2 – a system with a fragment of operators for the inverse transformation of data types transmitted from x_j to x_i after its execution;

S_0^1 – a piece of software structures with operators epilogue for the vertex x_i, for the restoration of the environment.

Для описанных структур программ зададим операции link для сборки отдельных программ на рис.8:

Link Prog P_1 (x_1, x_2);

Link Prog P_2 (x_1, x_3) (x_3, x_6);

Link Comp P_3 ((x_1, x_3) (x_3, x_5/ x'_{58})+ (x_5, x_7 (20)

Link Prog P_4 (x_1, x_4), (x_4, x_6);

Link Comp (P_1 U P_2 U P_3 U P_4).

The programs of the complex (aggregate, packets) are given unique names (P_1, P_2, P_3, P_4) corresponding to the root names of modules in the chains of the graph.

Thus, the process of constructing the program structure on the graph includes:

1. Enter the module description in the LP (L') and perform syntax checking.
2. Select the required modules and interfaces from the repositories and place them in the graph.
3. Translation of the unit modules in the LP.
4. Generation of communication modules for each interconnected pair of graph modules.
5. Assembly of the elements of the graph in the finished structure, linking modules in the operating system (IBM, MS, Oberon, Unix и др.) [1-5].
6. Test the system on data sets and assess the reliability of the unit.

After the modules are built, the name of the software Assembly is entered into the boot library. If you create a fragment that is later included in another aggregate, its name must match the name of the main module. In connection with the transition to the Internet environment to work with various software and system services in the configuration assembly of such tools provides security, data protection and quality assessment of ready-made modules, service resources and web systems in Internet.

3.1 Evaluation of the reliability and quality of systems

The key characteristics of quality attributes is reliability and completeness as properties of the AS to eliminate failures with hidden defects with this criterion and a quality model, which relates the measures and metrics of the internal, external and operational type. From the standpoint of completeness of the product is the main indicator of quality are defects and failures.

The model of defects based on multiple quality factors, analysis of causal relationships between them, combining qualitative and quantitative assessments of their impact on the density of defects [15-18, 22]. To calculate the *reliability function* uses a special formula:

$$R(t \mid T) = \exp(-(m(T+t) - m(T))),$$

Where *t* - the operating time of PS without a failure when testing in a period of time T; *m (T)* is a function of reliability growth, as the average number of defects.

The reliability of the software largely depends on the number remaining and corrected errors in the development process. During operation, errors are also detected and eliminated. If the bug fixes are not made new, or at least new bugs introduced is less than clear, in the course of operation reliability increases.

To assess the *quality* systems used the standard quality model:

M_{gua} = {Q, A, M, W}, where

Q = {$q_1, q_2, \ldots, q_i$} i = 1,..., 6, – various quality characteristics (Quality – Q);

A = { $a_1, a_2, \ldots, a_j$} j = 1,..., J, – the set of attributes (Attributes – A), each of which captures a separate property of the qi quality characteristics;

M = {$m_1, m_2, \ldots, m_k$} k =1,..., K, – the set of metrics (Metrics M) each element of the attribute aj for the measurement of this attribute.

W = {$w_1, w_2, \ldots, w_n$}, n = 1,..., N are weight coefficients (Weights W) for metrics of the set M.

The quality standard identifies six basic quality characteristics: q_1: functionality; q_2: reliability; q_3: use; q_4: efficiency; q_5: maintainable; q_6: portability.

The quality $q_1 - q_6$ are assessed by the general formula:

$$q_1 = \sum_{j=1}^{6} a_{1j} m_{1j} w_{1j}$$

On the basis of obtained quantitative characteristics of the final grade is calculated by summing the values of individual indicators and their comparison with the benchmark systems.

Configuration of ready-made software recourses in the system

Under the configuration of the system is understood the structure of some of its version, including software elements, combined with each other by link operations with parameters that specify the options for the functioning of the system [1-5, 16-22]. Version or system configuration according to the IEEE Standard 828-2012 (Configuration) includes:

– configuration basis – BC;
– configuration items;
– program elements (modules, components, services, etc.) included in the graph;

Configuration Management is to monitor the modification of configuration parameters and components of the system, as well as to conduct system monitoring, accounting and auditing of the system, maintaining the integrity and its performance. According to the standard, the configuration includes the following tasks:

1. Configuration identification.
2. Configuration Control.
3. Configuration Status Accounting.
4. Configuration audit.
5. Trace configuration changes during system maintenance and operation;
6. Verification of the configuration components and testing of the system.

A configuration build uses a system model and a set of out-of-the-box components that accumulate in the operating environment repositories or libraries, and selects their operating environment Configurator (for example, in http://7dragons.ru/ru). The Configuration assembles the components according to their interfaces [1-5, 9, 15] and generates a system configuration file.

The Configuration builds components and reuses for the AS using config operations to obtain the output PP file. The product must be tested on a set of tests with the least number of errors. After that, the resulting PP is evaluated for quality taking into account the incoming resources in the AS. For maintenance of PP the certificate of quality according to standards has to be issued.

4 The perspective technology for future

Coordinated and parallel programming provides a division of the computational process into several subtasks (processes) for TRAN's computers and supercomputers, the results of which are sent via communication channels. Languages for parallel programming - PVM, LAM. CHMP and MPI (Message Passing Interface) interface descriptions and OpenMP. The POSIX standard provides messaging between programs in LP of C, C+ and Fortran [15, 25].

Programming on classes, on a prototype in OOP and Object-component methods - OCM

OCM are the mathematical design of systems by Graph from ready-made resources (objects, components, services, etc.) to OM (Object Model). It is the formal method which transform the elements OM to a component model or a service model [15, 25, 32, 33]. http://0x1.tv/20181122AF.

Component paradigm. The basis of this paradigm - OCM graph in which vertexes graph are the components of the CRP (reuses), interfaces and arcs specify the subject classification and the relationship between the vertices. Components are described by the formalisms of the triangle of Frege [15, 29].

Service-component paradigm. System and service-components - web resources implement intellectual knowledge of specialists about applied fields in the Internet environment. Each implements some function and communicates with the technological interface to interact with other services through protocols and provide Assembly and solution of applications of different nature [18].

Methods of production of factories (Product Line/Product Family) programs and Appfab and certificate them of the quality are discussed [20, 23].

Application of the ontology language OWL (www.semantic_web.com), resource language (RDF) and intelligent agents of ISO 15926 standard for networking. Ontology of Life Cycle and Computational geometry is a part of computer graphics and algebra. Used in the practice of computing and control machines, numerical control etc. is also used in robotics (motion planning and pattern recognition tasks), geographic information systems [27].

The Agile methodology is focused on the close collaboration of a team of developers and users. It is based on a waterfall model lifecycle incremental and rapid response to changing demands on PP. Variants of Agile: eXtreme Programming (XP), SCRUM, DSDM…

Internet Technologies

Within the Internet (Things, Smart IoT) technologies are developing in the direction of creating smart computers, cities, robots and devices for use in medicine, genetics, physics, etc. The information objects (IO) that specifies the digital projection of real or abstract objects that use Semantic Web Ontology interoperability interfaces. IO through Web services began more than 10 years ago. Interaction semantics IO is based on RDF and OWL language of ISO 15926 Internet 3.0. The next step of the development of the Internet is Web 4.0, which allows network participants to communicate, using intelligent agents. A new stage in the development of enterprise solutions-cloud (PaaS, SaaS) who spliced with Internet space and used to create Adaptive applications. Cloud services interact through the Web page by using agents [28]. Internet stuff (Internet of Things, Smart IoT) indicates the Smart support competing APPS using distributed micro services such as Hyper cat (mobile communications); industrial Internet (Industrial), covering the new automation concepts-smart energy, transportation, appliances, industry», and another.

Computer nanotechnology

Today computer nanotechnology is actually already working with the smallest elements, "atoms" similar to the thickness of the thread (transistors, chips, crystals, etc.). For example, a video card from 3.5 million particles on single crystal, multi-touch maps for retinal embedded in the eyeglasses, etc.[28].

In the future, ready-made software elements will be developed in the direction of nanotechnology by "reducing" to look even smaller particles with predetermined functionality. Automation of communication, synthesis of such particles will give a new small element, which will be used like a chip in a small device for use in medicine, genetics, physics, etc.[28-33].

5 Conclusion

In this article proposed the theoretical apparatus of graphs to create modular program structures. The graph theory allows us to establish the shortest path of program elements (modules) and prove the correctness of binding graph modules using adjacency matrices, reach ability and mathematical operations (association, connection, difference, etc.) in complex program structures (complex, system, packets, etc.). Author are formulated the theoretical aspects of the application of graph theory in [6-15]. Since 2013, graph theory has been used in the modeling of complex systems of objects, components, services, etc. (OCM) [15] and has been used in the world practice in the transition to the Internet environment [6-14. 22-29]. The paper describes the features of modeling systems using graph theory and mathematical operations on elements of software structures with using Assembly operations (link, make, config, etc.), which are implemented in different environments. Module objects testing and controlled on quality. After config programs elements – modules should be verified and tested, then checked for quality. The AS made of them is checked according to standard ISO/IEC 9000 (1-4) for quality, and then transferred to the customer (Project RFFI N19-01-00206).

The graph theory, programming paradigms and ontology of mathematical modeling of applied problems for vital areas of society (medicine, biology, physics, mathematics, economics, etc.) will become the main tools of smart machines of the 21st century [15, 24-33].

Reference

[1] Lavrischeva E. M. , Grishchenko V. N. The connection of multi-language modules in the OS of the ES.- Moscow, 1982.- 127p.

[2] Lavrishcheva E. M. , Grishchenko V. N. Assembly programming. –K.: Of Sciences. Dumka.1991.-136p.

[3] Lavrishcheva E. M. , Grishchenko V. N. Assembly programming Basics of software industry products'. K.: of Sciences.Dumka.-2009.-371p.

[4] Glushkov V. M., Stogniy A. A., Lavrishcheva E. M. and others. System of automation of production of programs (The APROPOS) .-Kiev, 1976.-134p.

[5] Lipaev V. V., Posin B. A. ,Shtrik A. A. the Technology of Assembly programming.-M.: 1992.-284 p. 6.

[6] Rimsky G. V. Structure and functioning of the modular automation system programmings.- Artificial intelligence: application in chemistry.-1987.-№5.-p. 36-44.

[7] Halstead M. H. the beginnings of a science about the programs.- Perevod. with ang. –M.: Finance and statistics.- 1981.-201p.

[8] Horn, E., Winkler, F., Design of modular structures.– Computer technology of the socialist countries.- 1987.- Issue .21.-p. 64-72.

[9] Koval G. I., Korotun T. M., Lavrishcheva E. M. On one approach to solving the problem of intermodule And technological interface// All. the collection of the Academy of Sciences and Min.University of the USSR.-1987.

[10] Agafonov V. N. Program specification: conceptual tools and their organization. - Novosibirsk. - Science, 1987. - 380p.

[11] Kotov V. E., Introduction to the theory of program schemes, Novosibirsk, 1978.

[12] Nepeyvoda N. N. Program logic. - Programming, 1979, № 1, p. 15-25;

[13] Evstigneev A. N. Graph theory in programming, Moscow, Nauka. - 1985. -351p.

[14] Ershov, A. P., introduction to the theory of programming.-Moscow.-1977. - 287p.

[15] Lavrischeva E. M. The theory of object-component modeling of software systems. Preprint the Russian Academy of Sciences, No. 29, 2016 - M: 48 p. ISBN 078-5-91474-025-9.13;

[16] Lavrischeva E. M. Ryzhov A. G. Application the theory of General data types of ISO/IEC 11404 GDT Standard in relation to Big Data.- The conference "Actual problems in science and ways their development", 27 December 2016, http://euroasia-science.ru.- p. 99-110.

[17] Lavrischeva E. M., Mytulyn V. S., Kozin S. V., Ryzhov A. G. Creation of the application and information systems from ready-made Internet resources. The proceedings of ISP RAS.-M.: 2018, Volume 30. Issue.1 p.27- 40.

[18] Lavrischeva E. M. , A. G. Ryzhov. Approach to modeling systems and sites from ready-made resources.- XX All-Russian conference, September 17-22, 2018. Novorossiysk.-IPM im. M. V. Keldysh.- Report presentation. Publication in the collection.-p. 321-345.

[19] I. B. Burdonov, A. S. Kosachev, V. V. Kulyamin correspondence Theory for systems with locks and Destructions.- Moscow, 2008. - 411p.

[20] Lavrischeva E. M. Software Engineering of computer systems. Paradigms, technologies, CASE- means – Sciences. Dumka.- 2014.-284p.

[21] Bruno Courcelle, Joost Engelfriet Graph structure and monadic second-order logic. A language-theoretical approach (hal id: hal-oo646514) and Theory graph (wikipedia.ru, Foxford.ru).

[22] Lavrischeva E. M., Pakulin N.V., Ryjov A.G., Zelenov S. V. Analysis of methods of assessment reliability of equipment and systems. Practice of application of methods of reliability.-Scientific- practical conference - OS DAY, Moscow, 17-18th 2018. The proceedings of ISP RAS, том5 DOI: 10.15514/ISPRAS-2018-30(3), 2018. (http://0x1.tv/20180517F).

[23] Ekaterina M.Lavrischeva. Assemblling Paradigms of Programming in Software Engineering.- 2016, 9, p.296-317, http://www.scrip.org/journal/jsea, http://dx.do.org/10.4236/jsea.96021.
[24] E. M. Lavrischeva.The Scientific basis of software engineering.- International Journal of Applied And Natural Sciences (IJANS) ISSN(P): 2319-4014; ISSN(E): 2319-4022 Vol. 7, Issue 5, Aug - Sep. 2018; p. 15-32.
[25] Gorodnyay L. V. Programming Paradigms. Analysis of the state and prospects.-SORAN, 2018.-282p.
[26] Ekaterina Lavrischeva, Andrey Stenyashin, Andrii Kolesnyk. "Object-Component Development of Application and Systems. Theory and Practice". Journal of Software Engineering and Applications, 2014, http://www.scirp.org/journal/jsea.
[27] Lavrischeva Ekaterina. "Ontological Approach to the Formal Specification of the Standard Life Cycle", "Science and Information Conference-2015", Jule 28-30, London, UK, www.conference.thesai.org.- P.965-972.
[28] Lavrishcheva E.M. Petrov I.B. Ways of Development of Computer Technologies to Perspective Nano.- Future Technologies Conference (FTC), 29-30 November 2017| Vancouver, Canada-p.540-549.
[29] E. M. Lavrischeva. Component theory and collection of technologies for application development from ready-made resources/ / 4-Scientific and practical conference " Actual problems of system and software engineering.- 20-24 may 2015. Collection of scientific works APSE-2015.- pp. 101-119.
[30] E.M.Lavrischeva. Science of computer programs and systems in XX-XXI centuries: Past, Present, Future. European Journal Mathematics and Computer Science.- Vol.5 N 1.- p.67-87. 2018.-ISSN 2059-9951. www.idpublications.org.
[31] E.M. Lavrischeva. Scientific Basis of System Programming.- Journal of Software Engineering and Applications (JSEA), Vol. 11 No. 8 of August issue, 2018.-N 11.-p.408-434, ISSN online 1945-3124, ISSN Print 1945-3116.
[32] E. M. Lavrischeva. Software Engineering: New disciplines and e-learning them for development of Applied Systems. Europe Journal of Engineering and Technology, 2015.-N3. p. 36-67. ISSN2056-5860, www.idpublications.org.
[33] Lavrischeva E.M. (2017, IEEE). Development of the theory programs and systems in the USSR. History and modern Theory. - Sorucom-2017, IEEE Springer-2017.-p. 31-47.