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

E. M. Lavrischeva
Doctor of phys.-math. Sci., Professor of MIPT, scientifically specialist ISPRAS.
Lavrischeva@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 resources 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 \( M : X \rightarrow Y \).

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: \( CI = \sum_{i=1}^{n} K_{id} F(x_i) \), where \( K_{id} \) - the weight coefficient of the parameter; \( F(x_i) \) – the element function for the parameter \( x_i \).

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; \( \Gamma \) - a finite subset of the direct product of \( X \) 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; \( \chi \) - a characteristic function given on the set of vertices \( X \) of the graph \( G \).

The value of the characteristic function \( \chi \) 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 \( G_1 \equiv G_2 \) between sets \( Y_1 \) and \( Y_2 \) exists an isomorphism \( \phi \), and for any \( x \in X \) \( F_2(x) = \phi(f_1(x)) \).

1 Development on Project RFFI № 19-01-00206
Definition 2. Two program structures \(S_1 = (T_1, \chi_1)\) 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 \((G, \chi)\). 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 heterogenous 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 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 \(X \times X \times Z\) - arcs of the graph, corresponding to a finite vertex (Fig. 1).

The set of arcs of the graph have the form: \(E = \{(x_1, x_2, 1), (x_1, x_3, 1), (x_5, x_6, 1), (x_5, x_8, 2)\}\). 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 \(x_8\) there are two calling operators to the modules, with vertices \(x_7, x_8\).

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

\[\chi(x) = 1\] if the module corresponding to the vertex \(x \in X\) - included in the unit;
\[\chi(x) = 0\] if the module corresponding to the vertex \(x \in X\) - 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.

Function \(F\) maps \(X\) to \(Y\), \(F: X \rightarrow Y\) \hspace{1cm} (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](image)

A subgraph - a fragment of a software aggregate \( G' = (X', E') \) 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 \( \chi(x) = 0; \)

- \( R(S) = 0, \) 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: \( G_m = (X_m, E_m) \). It contains a single vertex \( x \in X_m \) for which \( \chi(x)=1 \). This vertex is the root. An arc of the form \((x_j, x_k)\) means calling the module to the corresponding vertex \( x_j \), i.e. to the module with the vertex \( x_k \). 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: \( G_c = G_1 \cup G_2 \cup ... \cup G_n \),

where \( X_c = X_1 \cup X_2 \cup ... \cup X_n \) and \( E_c = E_1 \cup E_2 \cup ... \cup E_n. \)

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 \( A \) (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:

\[
V = \begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{pmatrix}
\]

\[
M = \begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

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:

\[
V_i = \begin{pmatrix}
1 \\
0 \\
0 \\
\end{pmatrix},
\]

\[
M_i = \begin{pmatrix}
0 & 1 \\
0 & 0 \\
0 & 0 \\
\end{pmatrix};
\]

\[
V_i = \begin{pmatrix}
1 \\
0 \\
0 \\
\end{pmatrix}.
\]
The reachability graph of program structures

Let \( G = (X, E) \) be a graph of a program of modular structure; \( x_0, x_j \) - vertices belonging to \( X \). If there is an oriented chain from \( x_0 \) to \( x_j \) in the graph \( G \), then the vertex \( x_j \) is reachable from the vertex \( x_0 \). The following statement is true: if the vertex \( x_j \) is reachable from \( x_i \) and \( x_k \) from \( x_j \), then \( x_k \) is reachable from \( x_i \). The proof of this fact is obvious.

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

\[
\overline{x_i} = \{x_j\} \cup D(x_j)
\]

(5)

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 \( (x_1 \rightarrow \text{root vertex}) \).

\[
\overline{x_1} = \{x_1\} \cup D(x_1) = X.
\]

(6)

**Evidence.** Suppose the vertex \( x_0 \ (x_0 \in X) \) is unattainable from \( x_1 \). Then \( x_0 \notin \overline{x_1} \) 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 \( H(x_j, x_k) \) leading from \( x_j \) to \( x_k \). Based on the acyclicity of the graph \( G \), in \( X' \) there should be a simple chain \( H(x_s, x_k) \), where the vertex \( x_s \) does not include arcs (this chain can be empty if \( X' \) consists only of \( x_s \)). Consider the chain \( H(x_s, x_k) = H(x_s, x_l) \cup H(x_l, x_k) \). This means that the module \( x_l \) is reachable from vertices \( x_s \) and \( x_k \) 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 \( x_2 \), \( x_3 \), \( x_6 \) will never be executed.

Thus, the results of theorem 1 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).

\[
A = \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

(7)

Graph theory.
Therefore, the sequence of degrees of the adjacency matrix 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 \( \overline{M} = \sum_{i=1}^{n} M^i \), 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 \( \leq p \). Let the coefficient \( m_{ij}^{l} \) of the matrix \( M \) be different from zero. This means that there is a degree of matrix \( M^l \) in which the corresponding coefficient \( m_{ij}^{l} \) 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 \( M_i \), coinciding with the reachability matrix \( A_i \), 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 \( M^l \). Then there is a cycle in the original graph.

**Evidence.** Let \( m_{ii} > 0 \) for some \( i \). Therefore \( x_i \) 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 \( M' \) 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 \( M' \) 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 \( M' \) 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 \( x_i \), then the opposite is not true.

We introduce the notation \( x_i \triangleright x_j \) 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 \( X \):

\[ x_1 \triangleright x_2 \triangleright \ldots \triangleright x_n \]

where \( \triangleright \) denotes the reachability ratio.

Since \( X \) is finite, the chain breaks. The vertex \( x_n \) has no outgoing arcs, i.e. the element \( x_n \) 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 \( (U, \cap, \cup, +, -) \) 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: \( S = (G, \chi) \).

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** \( S = S_1 \cup S_2 \) 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:

\[ G = G_1 \oplus G_2, \quad X = X_1 \oplus X_2, \quad E_1 \oplus E_2, \]

where the symbol denotes a direct sum provided:

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

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:

\[ V_{1,2} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \quad M_{1,2} = \]

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 \( G_i \) and \( G_2 \) of program structures \( S_i \) and \( S_2 \), respectively. This operation

\[
S = C(S_i, S_2)
\]

which is execute if these structures meet the following conditions:

- set \( X' = X_i \cap X_2 \) not empty;
- vertex \( x_i \in X \) and \( \chi(x_i) = 0 \);
- \( D^* (x) \cap D (x) = 0 \) for every \( x \in X' \), where \( D^* (x) \in X_i \) and \( D (x) \in X_2 \);
- \( G = G_i \cup G_2 \) \( X = X_i \cup X_2 \), \( E = E_i \cup E_2 \).

(13)

The characteristic function \( \chi \) is satisfied under the condition:

\[
\chi(x) = \chi_1(x), \text{ if } x \in X_i \setminus X'; \\
\chi(x) = \max \{ \chi_2(x), \chi_3(x) \}, \text{ if } x \in X', \\
\chi(x) = \chi_2(x), \text{ if } x \in X_2 \setminus X'.
\]

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

The third condition prohibits the existence of cycles in the result graph. Indeed, if there is \( x_0 \in D^*(x) \cap D(x) \), then \( x_0 > x \) and \( x > x_0 \), 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 \( X_i \cap X_3 \neq 0 \).

Then the result of the \( S_1 + S_3 \) join operation is undefined.

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

\[
G(X, I) = X = x_i \cup E = \{(x_i, x_j, K) \mid x_i, x_j \in X \},
\]

and for the characteristic function \( \chi(x) = \chi_1(x) \), 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 \( x_i \in X_i \). 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 \( G_i \)). Formally, the difference operation of the program structure has the form:

\[
S = S_1 - P_{x_i} (S_1)
\]

(15)

and defined as follows

\[
G = (X, E), \quad X = (X_i \setminus x_i) \cup X' \\
E = \{(x_i, x_j, K) \mid x_i, x_j \in X \}
\]

(16)

where the set \( X' \) consists of such elements for which

\[
X' = \{x_j' \mid x_j \in X_i \setminus x_i \} \cup \{x_j' \mid x_i \in X_i \setminus x_j \} \cup \{x_j \}.
\]

(17)

Here, the characteristic function \( \chi \) is defined as:

\[
\chi(x) = \begin{cases} 
\chi_1(x), & \text{если } x \in X_i \setminus x_i; \\
\chi(x) = 0, & \text{если } x \in X'.
\end{cases}
\]

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

\[
S - P_{x_i} (S) + P_{x_i} (S) = S.
\]

(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 \( G_i \) will have the same properties. The root vertex \( G \) is the same as the root vertex \( G_i \). 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 $G^* \times \chi^*$, where $G^*$ and $\chi^*$ 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:

$$\Omega = \{U, \cap, /, +, -\}, P, C, R$$  \hspace{1cm} (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](image)

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:

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

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 $x_0, x_{n+1}, \ldots, x_{n+m}$, where $m$ is the number of end vertices. New arcs include $(x_0, x_1, 1), (x_1, x_{n+1}, 1), \ldots, (x_{n+m}, x_{n+m+1}, 1)$. 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 $x_0, x_n, x_{10}, x_{11}, x_{12}$. 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:

$$l_{0,n+m} = v_{01} + \ldots + v_{p,n+m} = v_0 + \frac{v_1 + v_2}{2} + \ldots + v_{n+1} + v_{n+2} + \frac{v_{n+3}}{2} = v_0 + v_1 + \ldots + v_p + v_{n+1} + \frac{v_{n+2}}{2} = v_0 + \frac{v_1 + v_2}{2} + v_{n+1} + \frac{v_{n+2}}{2} + \frac{v_{n+3}}{2}$$

This length $l_{0,n+m}$ will be equal to the sum of the memory modules for path $x_0, x_{n+m}$.

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 $m > 0$, the corresponding location will be $v_p$. The values $m = 0$ 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 $l_{0,n+m}$ will be equal to the sum of the memory modules for path $x_0, x_{n+m}$.

241
For \( k = 1 \) to \( n \) do
For \( I = 1 \) to \( n \) do
For \( j = 1 \) to \( n \) do

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

The most complex structure for the values \( V_{0}^{\min} \leq V_{0} \leq V_{0}^{\max} \) can be constructed by following the algorithms proposed in [4-7]. The qualitative dependence of \( V_{0} \) 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 \( V_{0}^{\max} \) and \( V_{0}^{\min} \).

**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 CALL. 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:

\[
M = \begin{bmatrix}
  x_{1} & x_{2} & x_{3} & x_{4} & x_{5} & x_{6} & x_{7} & x_{8} \\
  0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

(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 \( V_{d}^{\min} = V_{e} \).

If each dynamic component consists of one module, then the modified Floyd algorithm finds the maximum path and \( V_{d}^{\max} = V_{0}^{\min} \).

Due to the duplication of modules there is an increase in the main memory of the OS. 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.
3 Operations of assembling elements of graph $g$

Let the graph $G$ be represented by the set of modules $X = \{x_1, x_2, \ldots, x_m\}$, 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 $x'_i$ 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:

$$
E = \{(x_s, x_t, 1), (x_s, x_t, 1), (x_s, x_t, 2)\}
$$

The aggregate is given a unique name corresponding to the generated root module. For the graph $E = \{(x_s, x_t, 1)\}$ a fragment of operators providing a dynamic call will be formed in the communication module $x'_i$. For a pair of modules specified in Fig.8 vertices $x_s, x_t$, 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:

$$
x'_i = \text{S}_0 \ast (\text{S}_1 \times \text{S}_2) \ast (\text{S}_2 \times \text{S}_3) \ast \text{S}_3
$$

where $\text{S}_3$ is a fragment of the aggregate that defines the environment of $x_j$ module functioning;

$\text{S}_1$ – 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;

$\text{S}_2$ – a system with a fragment of operators for the inverse transformation of data types transmitted from $x_i$ to $x_j$ after its execution;

$\text{S}_3$ – a piece of software structures with operators epilogue for the vertex $x_j$, for the restoration of the environment.

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

1. Link Prog $P_1 (x_1, x_2)$;
2. Link Prog $P_2 (x_1, x_3) (x_1, x_4)$;
3. Link Comp $P_3 ((x_1, x_5) (x_1, x_6/ x_3/ x_8)+ (x_5, x_7)$ (20)
4. Link Prog $P_4 (x_1, x_5) (x_1, x_6)$;
5. Link Comp $P_1 \cup P_2 \cup P_3 \cup P_4$.

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:

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

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:

$$M_{ss} = \{Q, A, M, W\},$$

where

- $Q = \{q_1, q_2, ..., q_i \}_i = 1, ..., 6$ – various quality characteristics (Quality – Q);
- $A = \{a_1, a_2, ..., a_j \}_j = 1, ..., J$ – the set of attributes (Attributes – A), each of which captures a separate property of the qi quality characteristics;
- $M = \{m_1, m_2, ..., m_k \}_k = 1, ..., K$ – the set of metrics (Metrics M) each element of the attribute $a_j$ for the measurement of this attribute.
- $W = \{w_1, w_2, ..., w_n \}, n = 1, ..., N$ 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:

$$q_i = \sum_{j=1}^{n} a_{i,j} m_{1,j} w_{1,j}$$

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:

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 verticles. 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

[21] Bruno Courcelle, Joost Engelfriet Graph structure and monadic second-order logic. A language-theoretical approach ( hal id: hal-00646514) and Theory graph (wikipedia.ru, Foxford.ru).


