Software Engineering
2025, Vol. 11, No. 2, pp. 30-39
https://doi.org/10.11648/j.se.20251102.11

Research Article

Theoretical Foundations of Neural Network Integration of System Software Modules

Evgeniy Bryndin*

Scientific Department, Research Center Nature Informatic, Novosibirsk, Russia

Abstract

The theoretical foundations of neural network integration of system software modules lie in the field of combining formal methods for automation, optimization and management of development, integration and maintenance of complex software systems and systems engineering. The article proposes a formalism of operator schemes for constructing programs with deterministically connected modules for any class of algorithms. The structures of description and execution of programs are considered. In programs, there are data (information) and control transfer links between operators. When training neural networks, program structures indicate only the links that control their execution. Distributed links between inputs and outputs of sets of operators are associated with modules. Program operator modules are numbered. This numbering is preserved in program execution. Programs of modules can be sequential, parallel and sequential-parallel. The structure of programs with deterministically connected modules is considered. In the proposed formalism of operator schemes, the solvability of the problem of constructing programs with deterministically connected modules is proved. These fundamentals provide a theoretical basis for developing systems in which neural networks act as tools for complex automation and optimization of design, integration, and management of system software modules. In the future, the development of this concept will contribute to the creation of self-regulating, adaptive, and scalable system architectures.

Keywords

Program Structures, Operator Diagrams, Programs with Deterministic Linked Modules

1. Introduction

The development of science, engineering, technology and industry leads to an increase in the volume of information processed using supercomputers. The efficiency of processing large system programs on a supercomputer and their memory management depends on the organization of the computing process [1-3].

The means for studying the behavior of programs and their memory management are operator schemes. To study the behavior of programs with deterministically related modules, operator schemes with natural interpretation are used. Operator schemes with natural interpretation were used to prove the solvability of the problem of constructing programs with deterministically related modules.

The construction of programs with deterministically related modules is carried out by ensembles of intelligent agents based on the proposed theoretical foundations for each class of algorithms by optimizing them [4-6]. Constructing programs using ensembles of intelligent agents allows creating

*Corresponding author: bryndin15@yandex.ru (Evgeniy Bryndin)

Received: 1 September 2025; **Accepted:** 12 September 2025; **Published:** 9 October 2025

more stable, adaptive and efficient systems by combining several specialized agents. This is a structure in which several agents with different characteristics, knowledge and strategies interact or work sequentially to achieve a common goal. Each agent can specialize in certain tasks or aspects of the system, and their interaction ensures higher quality of decisions. Agents can have different data models, processing algorithms, and decision-making methods. Agents exchange information to coordinate actions. Opinion or result aggregation methods are used to select optimal solutions. The system can dynamically change the role of agents or add new ones depending on the situation. Providing an integrated approach to complex tasks: different agents can specialize in different aspects, such as data processing, planning, training. Using machine learning methods and neural networks inside agents to improve their efficiency. Using coordination protocols, such as contract networks or behavioral models.

Ensembles of intelligent decision-making agents with optimization competencies help to exponentially reduce the waiting time for the result of continuous processing of programs with deterministically connected modules on supercomputers with predictive memory management, in comparison with existing supercomputers processing programs with random memory management [7, 8].

2. Structures of Descriptions and Executions of Programs

The following graphic symbols are used to illustrate the structures of descriptions and executions of programs:

o — operator description;

I — inputs, outputs of operators, sets of operators;

— — connections between inputs and outputs of operators, sets of operators, as well as operators and sets of operators.

Definition 1. Program that has a sequential description and execution is called sequential.

Sequential description of program is made up of conditionally linked sequences of operator descriptions. Let us illustrate the structure of a sequential description of program with an example (Figure 1):

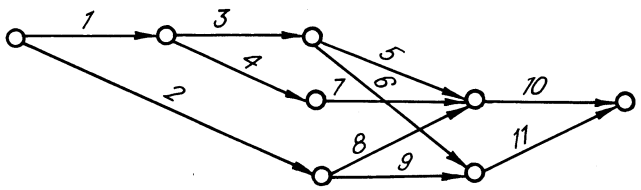

Figure 1. *Structure of a sequential program description.*

The program operators are executed sequentially.

Definition 2. Program that has parallel description and execution is called parallel.

Let us illustrate the structure of a parallel description of program with an example (Figure 2):

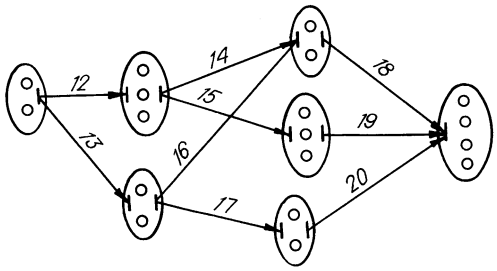

Figure 2. *Structure of a parallel program description.*

Operators combined in the description are executed in parallel. In parallel programs, operands are processed when they are ready.

Definition 3. Program that has a serial-parallel description and execution is called serial-parallel.

A serial-parallel description of a program is composed of conditionally connected sequences of sets of descriptions of combined operators and of conditionally connected sequences of descriptions of operators. Let us illustrate the structure of a serial-parallel description of a program using an example (Figure 3):

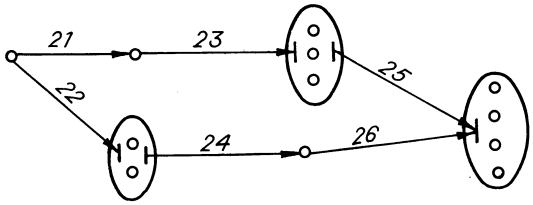

Figure 3. *Structure of serial-parallel program description.*

The execution of a program is sequential-parallel if at some moments the operators combined in the description are executed simultaneously, and at other moments only one operator is executed.

Definition 4. The executions of programs with different descriptions are called equivalent if they lead to the same output data for the same input data.

Definition 5. Programs with different descriptions are called identical in execution if they have the same number of equivalent executions.

An important characteristic of a program is the structure of links between operators. The structure of links in a sequential program can be linear, nonlinear, deterministic and nondeterministic. The structure of links in a parallel program can be linear, nonlinear, deterministic and nondeterministic. The structure of links in a sequential-parallel program can be linear, linear, nonlinear, deterministic and nondeterministic.

Definition 6. Programs with deterministic structure of module connections are called canonical.

3. Structure of Programs with Deterministically Linked Modules

Modules are formatted program structures containing processed and control information. A set of modules linked by data, algorithmically and by sequential execution is called a module-linked program. Let a module-linked program P be given, where

$$P = \{ PI_j \}, j = 1,\ldots, n.$$

Where PI_j is the operational module of the program, or data, or input-output.

$$PI_j = \{ UI_j, OD_j, LD_j, P_j, SPP_j, SPI_j \}$$

Where UI_j is control information,
OD_j is general data,
LD_j is local data,
P_j is the processing program,
SPP_j is the connection for searching for the module next in execution,
SPI_j is the algorithmic connection of modules for transferring execution.

Data links (information links) define relationships between modules for moving common data values. Moving common data values is defined by place pointers in modules and movement moments. Moments show the start of moving values, and place pointers in modules show where common data values are moving from and where. Movement pointers and moments can be static and dynamic. Information links between program modules indicate the route of moving common data values across their used modules.

Module search links define the direction of their processing. They can be static and dynamic. Dynamic links are defined by conditions that depend on the properties of algorithms, data values, and the influence of data on algorithm execution. SPP_j can be a program.

Search programs SPP_j process the values of variables that determine the connectivity of modules for algorithmic processing. Module search links determine the connectivity of each module with subsequent modules. These links are used when searching for subsequent modules for their proactive movement between external and RAM devices.

Execution transfer link programs SPI_j, which determine the algorithmic relationships of program modules, are specified either as static, or as an alternative depending on the order of module execution, or as a condition depending on data values. These links determine the input to the next module for processing.

Control information UI_j determines the module type, local and common data areas, addresses of dynamic link programs SPP_j and SPI_j, constants of static links SPP_j and SPI_j. It also determines asynchronous, parallel, conflict-free access to the processing module, movement of common data values, analysis of links between modules, movement of modules in memory and input-output processes, according to their activity priorities.

The sequence of modules $PI_1 \ldots PI_j \ldots PI_k$, connected via search programs $SPP_1 \ldots SPP_j \ldots SPP_k$, is called a modular execution PPI of program P, if the sequence of processing programs $P_1 \ldots P_j \ldots P_k$ determines the result of program P based on its data.

The execution of the processing program and the search program may depend on the values of the input modules, or they may be independent of them.

The information of the operational modules determines their processing, the movement of common data values between them. It is used to analyze the connectivity of modules, to search for them and move them around virtual memory.

Search programs, execution transfers, information links, and processing programs of different modules may have common data.

An important characteristic of a module-connected program is the structure of the links for searching for modules. Let the program P have a set $\{PPI^l\}$ of modular executions, where $l=1\ldots m$.

Definition 7. The link structure for searching for modules of a program P is deterministic if, based on its data, it is possible to execute sequences $\{(SPP_1 \ldots SPPj \ldots SPP_k)^l\}$ of module search programs for all modular executions $\{PPI^l\}$ of the program P.

Definition 8. A module-linked program P with a certain size of all modules is called a program with deterministically linked modules if the link structure for searching for its modules is deterministic.

In order to prove the possibility of constructing programs with deterministically linked modules for any class of equivalent algorithms, the formalism of operator schemes has been developed.

4. Operator Diagrams for Constructing Programs with Deterministically Linked Modules

The initial concept will be a graph. A graph in which the vertices represent operators, and the arcs represent possible transfers of control and data from one operator to another, forms the basis of operator diagrams.

Let program P be an enumerated list of elements $\{Oj\}$ that have execution transfer links, as well as information links between their inputs and outputs, where each input has an information link from some output. For a sequential program, each element Oj is an operator, for a sequential-parallel program it is either an operator or a set of parallel operators, and for a parallel program it is a set of parallel operators. In what follows, for the sake of uniformity of terminology, element Oj will be considered a generalized operator (either an operator or a set of parallel operators), and operators will not be

marked for now. When element *Oj* is a set of operators, it is assigned all the information links and execution transfer links of all operators in the set. Each chain of execution links corresponds to chains of information links.

The classes of sequential, sequential-parallel, and parallel programs intersect in the topology of links. Each operator has a certain number of arguments and results.

Operators have semantics {*Wj,k*}, according to which they transform arguments into results, where *j=1,...,n; k=1,...,m.*

Definition 9. A directed graph *G(X,G,U)*, in which the operators of a program *P* are represented by nodes $X = \{O_1, O_2, \ldots, O_n\}$ of the graph, and the information links by data are specified by the function $G: X \to Y$, and the links by execution between operators are specified by the function $U: X \to Y$, is called an operator diagram of the program *P*.

Operator diagrams have input and output control nodes. Control is transferred to the operator diagram through input nodes. Output nodes terminate the control of the operator diagram.

A graphical representation of a program in the form of an operator diagram can be transferred to a representation of algorithms if its actions are represented by nodes of the graph.

Definition 10. An operator diagram is called connected by control transfer if the graph *G (X, U)* is connected.

We will consider the execution of an operator to be the act from starting the processing of arguments to transferring their execution after receiving the result. We will agree that the transfer of the result to other operators occurs simultaneously with the transfer of execution.

The order of execution of operators in the operator diagram is specified by the marking of information links, links for transferring execution and operators. Different types of operators have their own markings and links. For further presentation of the material, we will consider operator diagrams with a possible order of operator execution (without marking).

Definition 11. Operator O_j is uniquely controlled by execution if it has one link for launching it for execution. Let us denote such a link for launching the operator for execution as *UZIj*.. The link *UZIj*.. is the input to the operator O_j, The uniquely controlled operator will be depicted in the operator diagram as a circle with an incoming arrow: ⟶ O

Definition 12. Operator O_j uniquely controls execution if it has one link for transferring execution.

Let us denote such a link for transferring execution by the operator O_j as *UPIj*. The link *UPIj*. is the output from the operator O_j.

We will depict the unambiguously controlling operator in the operator diagram as a circle with an outgoing arrow: ⟶ O

Definition 13. The operator O_j is alternatively controlled by execution if it has two or more connections for launching it for execution.

We will depict the alternatively controlled operator in the operator diagram as a circle with arrows entering it (Figure 4):

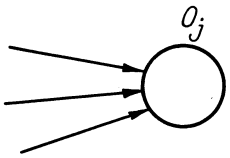

Figure 4. *Alternatively controlled operator.*

Definition 14. The operator O_j alternatively controls execution if it has several connections for the transfer of execution.

We will depict the alternative transfer of execution by the operator in the operator diagram as a circle with arrows coming out of it (Figure 5):

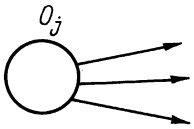

Figure 5. *Alternative transfer of execution by the operator.*

Definition 15. An operator O_j is called polysemantic if it has different execution semantics.

Let us denote the execution semantics of a polysemantic operator O_j by {*Wj,k*}. Thus, we move from an interpreted scheme to a scheme over a class of interpretations.

Definition 16. The operator O_j is called polyinformational if it has several information links for transferring the result as operands (arguments) to other operators. We will depict a polyinformational operator in the operator diagram as a circle with double arrows coming out of it, the sources of information links will be marked with bold dots. Information links coming from one source merge into one bundle.

Definition 17. The operator diagram *G(X,Y,U)* is called linear in execution if all operators in the diagram are uniquely controlled by execution and uniquely control execution.

A linear operator diagram graphically looks like this (Figure 6):

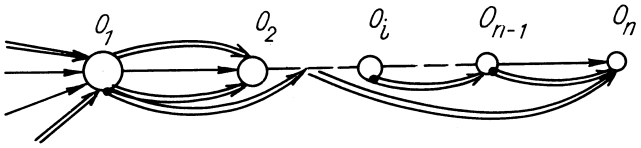

Figure 6. *Linear operator scheme.*

Definition 18. An operator scheme *G(X,Y,U)*, consisting of independent linear operator subschemes, is called a linear-type operator scheme in execution.

Each linear subscheme of a linear-type scheme has its own set of input data values. A linear-type operator scheme looks graphically as follows (Figure 7):

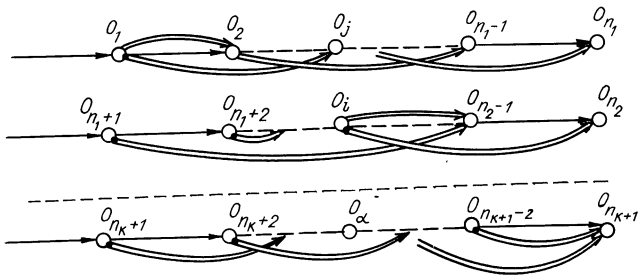

Figure 7. *Linear-like operator scheme in execution.*

The operator through which control is transferred to the subcircuit will be called the input, and the operator through which the subcircuit transfers control will be called the output.

Definition 19. An operator subcircuit, that has one input operator through execution links, alternatively controlling two or more operators of the subcircuit, and one output operator from the subcircuit through execution links, alternatively controlled by two or more operators of the subcircuit, where the input and output operators are different, is called a hammock-shaped operator subcircuit in execution.

For example, the following operator subcircuit is hammock-shaped (Figure 8):

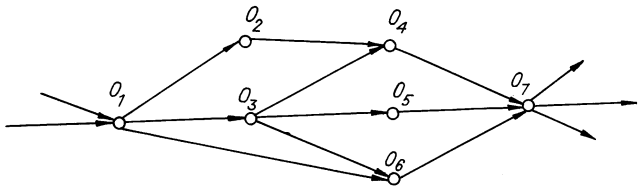

Figure 8. *Hammock-shaped operator subcircuit in execution.*

In this hammock-shaped subcircuit, the alternatively controlled first operator O_1 is the input operator to it through the execution links, and the alternatively controlled seventh operator O_7 is the output operator from it through the execution links.

Definition 20. The operator subcircuit $G((X_0 \subset X), \Gamma, U)$, in which the operators $X_0 = \{O_j\}$ form one closed chain $(Oj_1, Oj_2, \ldots, Oj_n)$ through the execution links $U: X_0 \to X_0$, is called a non-linked operator subcircuit with return on execution.

The operator subcircuit with returns graphically looks like this (Figure 9):

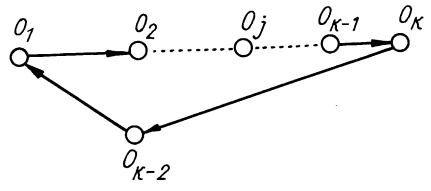

Figure 9. *Operator subcircuit with returns.*

Definition 21. Operator subcircuits $G((X_1 \subset X), Y, U)$ and $G((X_2 \subset X), \Gamma, U)$ with return on execution are called concatenated if the subcircuits have common operators $X_1 \cap X_2 = \varnothing$.

For example, the following operator circuit consists of concatenated subcircuits with returns (Figure 10):

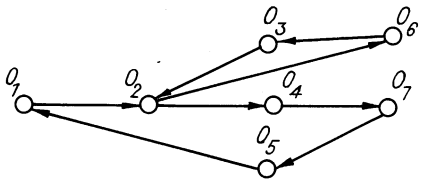

Figure 10. *Linked subcircuits with returns.*

Vertex O_2 is a common vertex of linked subcircuits with returns.

Definition 22. A nonlinear operator circuit G(X,G,U) without hammock-shaped subcircuits by execution and without subcircuits with return by execution is called a linearly cross operator circuit by execution.

For example, the following operator subcircuit is linearly cross (Figure 11):

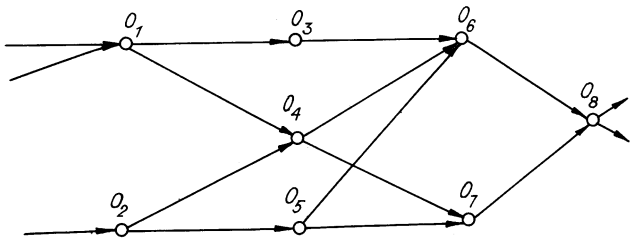

Figure 11. *Linear-cross operator subcircuit.*

In the proposed formalism of operator schemes with natural interpretation, the following theorems are proved.

Theorem 1. In the operator scheme *G(X,Y, U),* all hammock-shaped operator subschemes can be distinguished by execution.

Proof.

For each operator O_{j0} of the operator scheme *G(X,Y,U),* we select all operator routes $\{(O_{j0} \ldots O_{jn})\}$ via execution links. Consider all possible intersections of the selected routes. If the intersection of routes without the vertex O_{j0} is not empty, then they form a hammock-shaped subcircuit of the scheme *G(X,Y,U).*

Which is what was to be proved.

Theorem 2. In the operator scheme *G(X,Y,U),* we can select all linked and non-linked operator subcircuits with return.

Proof.

In the operator scheme, for each vertex, routes with a repeating vertex are selected. These routes are operator sub-

circuits with return. Subcircuits with return on execution that have common vertices are linked.

Which is what was to be proved.

Theorem 3. In the linearly crossed operator scheme *G(X,Y,U),* all linear subschemes can be distinguished.

Proof.

In the original operator scheme *G(X,Y,U),* we distinguish operators *{Oj(UZIj)},* operator schemes alternatively controlled by execution, and operators *{Oj(UZIj)},* alternatively controlling the execution of the scheme operators, as follows.

For each operator O_j of the operator scheme *G(X,Y,U),* we calculate the functions $U{:}O_j \to X$ and $U^{-1}{:}\ O_j \to X$.. An operator O_j for which the set $U^{-1}(Oj)$ consists of more than one operator is an alternatively controlled by execution operator *{Oj (UZIj),* and an operator O_j for which the set $U(O_j)$ consists of more than one operator is an alternatively controlling operator *{Oj(UPIj)}*..

In the operator circuit *G(X,Y,U),* we replace the operators *{Oj (UZIj),* alternatively controlled by execution and the operators *{Oj (UPIj)}* alternatively controlling execution with several unambiguously controlling and unambiguously controlled operators. We obtain subcircuits *{G(Xi,Y,U)},* consisting only of unambiguously controlled and unambiguously controlling operators. Consequently, the subcircuits *{G(Xi,Y,U)},* are linear operator subcircuits.

Which is what was to be proved.

4.1. Transformation of Operator Circuits by an Algorithmic Basis

We will consider the sequence of operators $O_1 \ldots O_i \ldots O_n$ of the operator circuit *G(X,Y,U)* to be its execution if each operator in the sequence is connected by control transfer and by information links only with subsequent (not necessarily adjacent) operators of the sequence.

Definition 23. The execution $Oi_1, Oi_2, \ldots, Oi_n$ of the operator circuit $G_1(X_1,Y_1,U_1)$ and the execution $Oj_1,Oj_2,\ldots,Oj_n$ of the operator circuit $G_2\,(X_2,Y_2,U_2)$ are called equivalent if for the same input data they lead to the same output data for natural interpretations of operator circuits as abstract programs and algorithms.

Definition 24. The operator circuits $G_1(X_1,Y_1,U_1)$ and $G_2\,(X_2,Y_2,U_2)$ are called equivalent if the sets of their executions are equivalent.

Definition 25. The programs P_1 and P_2 that have equivalent operator circuits are called equivalent.

Definition 26. The transformation of the operator circuit of program P_1 into the operator circuit of the equivalent program P_2 is called the transformation of operator circuits by the algorithmic basis.

Let us introduce two rules of tolerant transformation of synthesis and splitting of operator schemes by basis.

Definition 27. Transformation of operator scheme *G(X,Y,U),* in which polysemantic polyinformation operator is obtained from subscheme $G(X_1,Y,U)$ of operator scheme *G(X,Y,U)* with preservation of all information links and links on transfer of execution with operator scheme *G(X,Γ,U),* we will call synthesis.

We will demonstrate synthesis of operator subscheme on the following example (Figure 12):

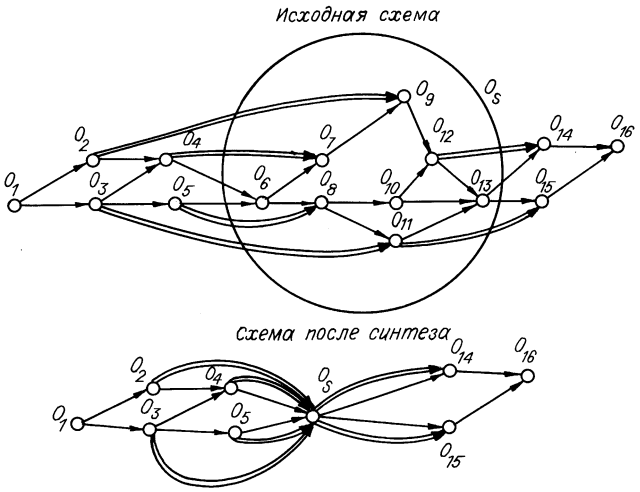

Figure 12. *Synthesis of an operator subcircuit.*

The information links and operands of the synthesized operator Os are determined by the information links of the operators of the equivalent subcircuit with other operators of the circuit. The semantics of the operator Os is equivalent to the functional semantics of the equivalent subcircuit.

Definition 28. The transformation of an operator subcircuit into several independent subcircuits with the preservation of all information links and links for transferring control with the operator circuit is called splitting.

We will demonstrate the splitting of subcircuits of an operator circuit into independent ones using a simple example (Figure 13):

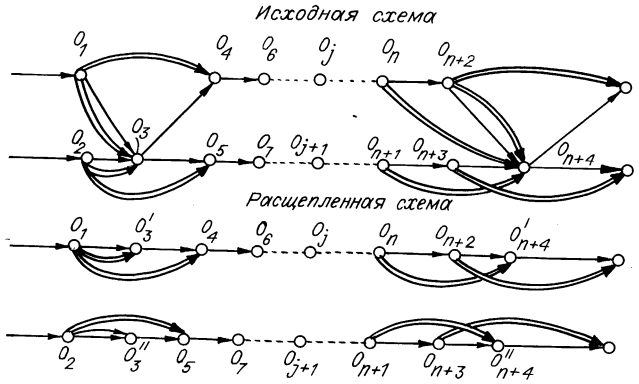

Figure 13. *Splitting of subcircuits of an operator circuit.*

In the resulting subcircuits, only the necessary information links and operands remain, and only those links for the

transfer of execution in these subcircuits that connect the operators of the subcircuits are preserved. Synthesis and splitting are the rules for transforming operator circuits according to the algorithmic basis.

4.2. Synthesis of Linear-cross Operator Circuits from Nonlinear Circuits

Theorem 4. An equivalent linearly cross operator scheme by execution can be synthesized from the operator scheme *G(X,Y,U)* with hammock-shaped operator subschemes by execution and with operator subschemes with returns by execution, linked and not linked.

Proof.

We synthesize polyinformation polysemantic operators $\{O_i\}$ from each hammock-shaped operator subscheme $G(X_i,Y,U)$.

We synthesize polyinformation polysemantic operators $\{O_i\}$ from each linked and not linked operator subscheme $G(X_j,Y,U)$.

In the original operator scheme *G(X,Y,U)*, all hammock-shaped subschemas $G(X_i,Y,U)$, concatenated and unconcatenated subschemas $G(X_j,Y,U)$ with returns on execution are replaced by equivalent polysemantic polyinformational operators $\{O_i\}$ and $\{O_j\}$.

We obtain an operator scheme $G(X_0,Y,U)$ without hammock-shaped subschemas and subschemas with returns, equivalent to the original operator scheme *G(X,Y,U)*. The resulting operator scheme will be linearly intersecting.

Which was required to prove.

4.3. Transition to Schemes Without Subschemes with Returns and Without Hammock-shaped Subschemas

Let *G(X,F,U)* be an operator scheme. We select all cyclic routes in the operator scheme by marking the vertices [1].

Let us consider paired intersections of cyclic routes. Non-empty intersections determine the concatenation of cyclic routes.

Let us take intersections of linked cyclic routes. Non-empty intersections define connectivity of linked cyclic routes. Connected cyclic routes will be linked subcircuits with returns.

Let us select all fully connected linked subcircuits with returns by intersection of cyclic routes. Let us represent a fully connected linked subcircuit with returns as a sequence of cyclic routes as follows. From the set of cyclic routes, let us select a route as the first element of the sequence, then, as subsequent elements of the sequence, let us put routes linked with the routes chosen subsequently and different from them. We select elements of the sequence until there are no routes in the set of cyclic routes linked with elements of the sequence and different from them.

Let us synthesize polyinformation polysemantic operators from linked subcircuits with returns. Let us replace the linked subcircuits with returns in the graph *G(X,T,U)* with equivalent operators obtained after synthesis with preservation of all information and execution transfer connections, with the remaining operators of the original scheme. We obtain the graph *G(X,F,U)* without linked subcircuits with returns.

Let, for example, *G(X,T,U)* be the operator circuit shown below, without subcircuits with returns. Let us distinguish in the operator circuit *G(X,T,U)* without subcircuits with returns the hammock-shaped subcircuits (Figure 14):

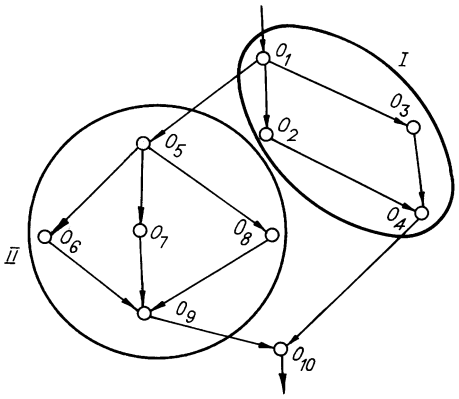

Figure 14. *Hammock-shaped subcircuits of the operator circuit.*

We synthesize polysemantic polyinformation operators from hammock-shaped subschemes. We replace hammock-shaped subschemes with vertices, preserving all connections with subschemes.

We obtain an operator scheme without hammock-shaped subschemes (Figure 15):

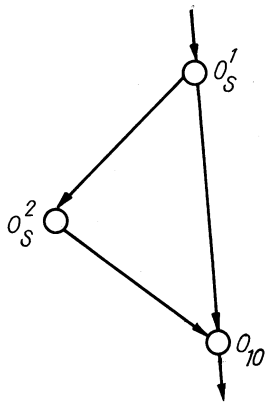

Figure 15. *Operator diagram without hammock-shaped subcircuits.*

4.4. Splitting of Linear-cross Operator Schemes into Linear-like Ones

Theorem 5. A linear-cross operator scheme G(X,Y,U) can be split into an equivalent linear-like operator scheme $G_1(X_1,Y_1,U_1)$.

Proof.

In a linear-cross operator scheme, we split the operators alternatively controlled by execution and the operators alternatively controlling execution into operators uniquely controlled by execution and uniquely controlling execution, preserving all information links and by execution with the corresponding operators.

We replace the operators alternatively controlled by execution with equivalent operators uniquely controlled by execution, and the operators alternatively controlling execution with equivalent operators uniquely controlling execution.

We obtain linear $G(X_j,Y,U)$ subschemes that are not connected with each other either by information or by execution. Linear subcircuits $G(X_j,Y,U)$ form a linear-shaped circuit equivalent to the original linear-cross circuit $G(X,Y,U)$.

Which is what was to be proved.

4.5. Construction of Programs with Deterministically Connected Modules.

Definition 29. A subcircuit of an operator circuit that processes a group of input data independently of an additional subcircuit is called autonomous in input data.

Theorem 6. Any operator circuit defined in one algorithmic basis can be transformed by synthesis and splitting into an equivalent one defined in another algorithmic basis and consisting of subcircuits autonomous in input data.

Proof.

Any operator circuit can be transformed into a linear-shaped one. A linear-shaped circuit consists of subcircuits autonomous in input data.

Which is to be proved.

Theorem 7. Any program can be transformed into an equivalent program with deterministically connected modules.

Proof.

Let P be the original program. We transform the program P into an equivalent program with a linearly shaped operator scheme. The resulting program has operator subschemes that are autonomous with respect to the input data. We divide the resulting program into modules. We obtain a module-canonically connected program. The resulting program will be equivalent to the original program.

Which was required to be proved.

In the proposed formalism of operator schemes with natural interpretation, the solvability of the problem of constructing programs with deterministically connected modules is proven.

Examples of constructing programs with deterministic coupled modules for neurocomplexation are considered in published works [1, 2] Programs with deterministic coupled modules can be obtained from programs with non-deterministic coupled modules by constructing and transforming their operator schemes into linear-cross schemes at the stage of neurocode generation. For programs with operator subschemes with returns and with hammock-shaped subschemes at the stage of neurocode generation, polyinformation polysemantic operators are formed in the form of software implementation by ensembles of intelligent decision-making agents based on the proposed theoretical foundations.

The proposed approach made it possible to continuously process large programs with deterministic coupled modules on the virtual memory of the supercomputer interpreter with preemptive replacement of modules on the virtual memory [2]. The interpreter command system is oriented towards the implementation of programs with linear-cross schemes.

5. Generating an Operator Diagram Based on a Description of the Functionality of System Modules

Generating an operator diagram based on a description of the functionality of system modules in natural language is the process of automatically converting a text description of requirements or functional specifications into a graphical or structured operator diagram. This approach significantly simplifies the design of digital systems, especially in the early stages of development, allowing you to quickly get a visual representation of the structure of the system and its components. The stages of this process include:

1) Identifying key elements: inputs, outputs, logical operators, conditions and sequences of actions.
2) Recognizing terms and concepts related to the functionality of the system.
3) Defining the logical sequence and relationship between elements that must be executed, in what order, under what conditions.
4) Identifying control elements (conditions, cycles, choices).

Automatic generation of an operator diagram includes:

1) Building a graph or block diagram, where the nodes are logical and control elements (conditional blocks), and the links are data flows.
2) Using templates or transformation rules based on typical structures of operator diagrams.
3) Displaying the resulting diagram for verification and subsequent editing.
4) Ability to export the diagram to formats suitable for further design automation.
5) Using models to analyze text and extract structured information.
6) Using a set of rules to transform semantic information into a graphical diagram.
7) Using machine learning to improve understanding of complex descriptions and recognize patterns.
8) Integration with CAD systems, UML editors or specialized platforms for automatic diagram generation.

This strategy allows neural networks to understand, interpret and interact with visual information in the same way as

people do. Using deep learning methods to train models makes them capable of understanding complex operator diagrams. Training models to associate operator diagrams with context is important for making code generation decisions.

6. Generating Module Code from Operator Diagrams

Generating module code from operator diagrams is the process of automatically converting graphical or formal descriptions of circuits into program code that implements the logic of these circuits [9-13]. Generating code from operator diagrams includes automatic transformation of an operator diagram into structured code. Code generation requires:

1) Creating code templates for each type of circuit element.
2) Automatic code assembly taking into account execution order, synchronization, and relationships between modules.
3) Inserting comments and metadata to facilitate understanding and further maintenance.
4) Optimizing the resulting code to improve performance or reduce resources.
5) Verifying the correctness of the generated code using simulations or formal methods.

Tools and methods used for automatic code generation from operator diagrams include specialized CAD systems, graph-to-HDL conversion systems, and the use of modeling languages such as SystemC or UML diagrams to describe circuits. This ensures that the design meets the desired specifications.

7. Conclusion

The areas of application of neural network system integration of software modules using the proposed theoretical foundations are:

1) Distributed control systems, robotics, autonomous vehicles.
2) Intelligent decision support systems.
3) Big data processing and analytics.
4) Automatic optimization of the composition and role of agents using machine learning methods. - Integration with neural network models to improve the perception and processing of information.
5) Development of standard platforms and protocols for the interaction of intelligent agents.
6) Creation of self-organizing ensembles capable of adapting to changes in the environment.
7) Ensuring compatibility and standardization of interactions [14].
8) Managing system complexity with an increase in the number of agents.
9) Ensuring security and reliability in distributed ensembles.

In general, building programs based on ensembles of intelligent agents opens up broad opportunities for creating flexible, scalable, and resilient systems that can effectively solve complex problems in various fields.

The future of neural network system programming promises significant changes in approaches to the development, automation, and management of software systems. Generating and optimizing system components using neural networks based on high-level specifications. Intelligent system design. Using AI to model complex system interactions and behavior. Creating high-level abstractions and automatically transforming them into effective implementations. Creating systems that can independently adapt to new threats and updates. Integration with DevOps and automated infrastructure management. Implementing neural network solutions in CI/CD processes, automatic scaling, monitoring, and resource management. Developing models that are trained on specific projects and quickly adapt to new requirements. Ensuring transparency, explainability, and compliance with ethical standards in automatic code generation and modification.

Overall, the future of neural network systems programming is associated with the strengthening of the role of AI in the automation of complex tasks, increasing the efficiency of development, and creating more stable, secure systems [15-18]. These technologies will be integrated into all stages of the software systems life cycle, transforming the software development industry.

Abbreviations

AI	Artificial Intelligence
CAD	Computer Aided Design
CI	Continuous Integration
CD	Compact Disc
HDL	Hardware Description Language
UML	Unified Modeling Language

Author Contributions

Evgeniy Bryndin is the sole author. The author read and approved the final manuscript.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] *Alexander Kirichenko*. Neural Network Programming. Neurocomputing Toolkit. *Created in the Intelligent Publishing System* Ridero. 2020.

[2] Zhengqing Liu, Musa Unal, Matthew J. Parkinson, Marios Kogias. DORADD: Deterministic Parallel Execution in the Era of Microsecond-Scale Computing. ACM. 2025. *Vivekkothari*. Difference between Deterministic and Nondeterministic Algorithms. 2022.

[3] *Balder ten Cate, Tobias Kappé.* Algebras for Deterministic Computation are Inherently Incomplete. POPL 2025.

[4] Evgeny Bryndin. Intellectual Agent Ensemble with Professional Competencies, Pattern Recognition and Decision Making Applied Science and Innovative Research, Vol 6, No 4 (2022). pp. 1-10.

[5] Evgeny Bryndin. Ensembles of Intelligent Agents with Expanding Communication Abilities. Acta Scientific Computer Sciences 5.2 (2023): 44-49.

[6] Evgeny Bryndin. Creation of Multi-purpose Intelligent Multimodal Self-Organizing Safe Robotic Ensembles Agents with AGI and cognitive control. COJ Robotics & Artificial Intelligence (COJRA). *Vol. 3, Issue 5. 2024.* pp. 1-10. https://doi.org/10.31031/COJRA.2024.03.000573

[7] Evgeniy Bryndin. Construction and Continuous Processing of Programs with Determinate Connected Modules. International academic journal *Electrical and Computer Engineering*, Volume 1, Issue 1. 2017, pp. 1-8.

[8] Evgeniy Bryndin. Supercomputer BEG with Artificial Intelligence of Optimal Resource Use and Management by Continuous Processing of Large Programs. International Journal of Research in Engineering, Vol. 1, Issue 2, 2019. Pages: 9-14.

[9] *H. Peter Alesso.* Vibe Coding by Example. *Independently published.* 2025. 329 p.

[10] Greg Lim. Vibe Coding for Beginners with Python and ChatGPT. Independently published. 2025.

[11] *David Gillette*. Vibe Coding in Python: The Python Programmers Guide to AI-Powered Programming (Generative AI Mastery). *Independently published.* 2025. 171 p.

[12] *Addy Osmani*. Vibe Coding The Future of Programming: Leveraging Your Experience in the Age of AI-Assisted Coding (First Early Release). O'Reilly Media, Inc. 2025. 250 p.

[13] *Addy Osmani*. Beyond Vibe Coding. Publisher(s): O'Reilly Media, Inc. 2025.

[14] Evgeny Bryndin. International Standardization Safe to Use of Artificial Intelligence. *Research on Intelligent Manufacturing and Assembly*. Volume 4, *Issue 1. 2025. pp. 185-191.*

[15] Evgeny Bryndin. Formation of Motivated Adaptive Artificial Intelligence for Digital Generation of Information and Technological Actions. *Research on Intelligent Manufacturing and Assembly*. V. 4, Is. 1. 2025. *pp. 192-199.* https://doi.org/10.25082/RIMA.2025.01.006

[16] Evgeny Bryndin. Network Training by Generative AI Assistant of Personal Adaptive Ethical Semantic and Active Ontology. International Journal of Intelligent Information Systems Volume. 14, Is. 2. 2025. pp. 20-25. https://doi.org/10.11648/j.ijiis.20251402.11

[17] Evgeny Bryndin. Self-learning AI in Educational Research and Other Fields. *Research on Intelligent Manufacturing and Assembly*. V. 3, *Issue 1. 2025. pp. 129-137.*

[18] Evgeny Bryndin. Technological Stages of Neural Network AI Generation of System Program Code Based on Modular Neuro Integration. American Journal of Embedded Systems and Applications 2025. *In press.*