Specifics of Semantics of a Statically Typed Language of Functional and Dataflow Parallel Programming

Alexander Legalov[1[0000-0002-5487-0699]], Igor Legalov[1[0000-0003-2630-5315]]
Ivan Matkovskii[1,2[0000-0002-4801-7982]]

[1] Siberian Federal University, 79, Svobodny pr., 660041, Krasnoyarsk, Russia
legalov@mail.ru

Abstract. It is proposed to add a static system of types to the dataflow functional model of parallel computing and the dataflow functional parallel programming language developed on its basis. The use of static typing increases the possibility of transforming dataflow functional parallel programs into programs running on modern parallel computing systems. Language constructions are proposed. Their syntax and semantics are described. It is noted that the need to use the single assignment principle in the formation of data storages of a particular type. The features of instrumental support of the proposed approach are considered.

Keywords: Programming Paradigms, Parallel Programming, Function And Dataflow Parallel Programming, Static Typing, Parallel Computing Models, Polymorphism.

1 Introduction

Modern methods of developing parallel programs are highly dependent on the features of architectures of parallel computing systems (PCS), which is reflected in programming languages. Almost any changes in the architecture of the PCS lead to the rewriting and modification of the already developed and debugged code. An attempt to overcome this situation is the application of the concept of architecture-independent parallel programming (AIPP), focused on the development of programs using language and tools designed for abstract (virtual) parallel systems with unlimited computing resources and dataflow strategies for managing by calculations. Such approaches are developing in different directions. We can mention the COLAMO programming language developed for systems on a chip [1, 2]. The creation of universal languages that are not directly related to architectural restrictions can be traced on the example of the functional parallel programming languages Sisal [3] and Pifagor [4].

The most consistent concept of AIPP was reflected in the Pifagor programming language and it is directly taken into account in its model of dataflow functional parallel computing. The program model is described as a resource-unlimited acyclic unconditional graph in which control is carried out according to data availability. In addition, it implements the principle of the single usage of computing resources [4]. At the model level, it is assumed that for carrying out any operations unique resources are allocated, the real distribution of which is carried out after the logical structure of the program is developed and debugged. To test the capabilities of the language, tools have been developed that support the process of creating, converting, and executing dataflow functional parallel programs [5].

However, the low efficiency of program execution should be noted, because of the use of an interpreter. This is due to the fact that the language uses dynamic typing of data, and the operators presented in the calculation model have dynamic behavior, allowing to create lists of arbitrary dimensions during calculations. In this regard, it is practically impossible to efficiently transform written programs into modern statically typed languages used in real parallel programming.

At the same time, experiments conducted using the developed tools showed the possibility of effective application of this paradigm for optimization [6], formal verification [7], and debugging [8] of programs even before their transformation to a specific architecture begins. This allows to have a program, the transfer of which to real PCS could be carried out more formally by imposing resource constraints that take into account the specific architecture, while preserving the already fine-tuned general logic of functioning.

In this regard, the modification of the dataflow functional model of parallel computing (DFMPC) is seen as promising and aimed at taking into account the features of data organization in modern programming languages, which would simplify the process of transforming dataflow functional parallel (DFP) programs. Basically, this modification is associated with the use of static typing and fixing the dimensions of list and container data structures, which leads to a revision of a number of concepts of DFMPC. In accordance with these changes, the DFP programming language should also change.

As a result of the research, a statically typed model of dataflow functional parallel computing (STMDFPC) was formed. Like the preceding DFMPC, it defines the program as an information graph with data flow control. However, the operators describing the program algorithm are developed taking into account possible transformations into statically typed programming languages, which leads to a change in a number of axioms and transformation algebra. Based on the proposed model, a statically typed dataflow functional parallel programming language Smile is developed.

2 Static Typing at the Operator Level

As in the previous DFMoPC [4], the operators specify the nodes of the information graph in which the calculations are performed according to the readiness of the data. However, there are a number of features associated with changing requirements. We must provide support for the following properties specific to statically typed programming languages:

- efficient transformation of statically typed dataflow functional parallel programs into other computation models instead of their interpretation;

- control is increased through the use of strong typing;
- to maintain the principle of dataflow control and the general concept of a dataflow functional model of parallel computing;
- each of the program-forming operators should rely on typed data controlled at the compilation stage;
- container (list) data types must have a fixed size, determined either at compile time or at run time;
- the language axiomatic should be simplified to reduce the number of dynamic checks and transformations at runtime;
- simplification of the algebra of equivalent transformations.

The above requirements lead to a change in almost all the DFMPC operators, as a result of which a calculation model with other properties is formed. These properties are determined through the features of the functioning of the program-forming operators of the STMDFPC.

The interpretation operator describes the functional transformations of the argument. It has two inputs to which the function F and the argument X arrive through the information arcs. Both the argument and the function can be the results of previous calculations. The main features of the new version of this operator are:

- types of arguments on operator inputs must be known at compile time;
- the type of output result is also computed at compile time;
- at the input and output of the operator, named data types, structures and tuples are allowed;
- for named types, only named equivalence is allowed;
- for tuples, structural equivalence is allowed;
- all basic operations must been predeterminated and their possible data types of arguments and results are fixed at the language level.

Based on this, signatures specifying the types of arguments and results are defined for the basic functions of the language. For user-defined functions, the types of arguments and results are explicitly specified during function definitions. The dualism of some basic data is allowed, which, depending on the use in the interpretation operator, can act either as an argument or as a function. In this case, it is possible for them to define a double type made of data type and function signature

The interpretation operator is launched when the data is ready, which is fixated by the appearance of markup on the input arcs. The result is set by marking the output arc.

Instead of grouping into a list of data in STMDFPC, grouping in a tuple is used. The following main properties of this operator can be distinguished:

- the size of the tuple is determined at compile time (due to the necessity to know the types of grouped data and their size);
- tuple elements are data of named types;
- comparison for structural equivalence with other tuples is provided;
- the readiness of the tuple for execution is determined by the readiness of all its data;

- there are no internal equivalent transformations that change the size of the tuple at runtime (the signal that is removed from the list in the DFMPC is a data type without a value and is stored explicitly).

The axioms that determine the transformation of tuples during calculations are also changed, which is also due to the introduction of additional control during compilation.

Grouping in parallel lists is replaced by grouping in a swarm. It is used to combine data over which one large-scale operation is performed. Swarm properties include:

- swarm size is determined at compile time;
- swarm elements are data of one named type or all swarm elements are structurally equivalent;
- the readiness of the swarm for execution is determined by the readiness of at least one element (asynchrony in the processing of its individual elements);
- there are no internal equivalent transformations that change the size of the swarm at run time;
- inside the tuples, the swarm does not degenerate into a sequence of elements of the tuple, but is a single element;
- the algebra of equivalent swarm transformations is implemented only at compile time.

The above characteristics make it possible to consider the swarm as a set of independent data that is launched as they become available. A swarm consisting of elements of the same type is also formed at the output of the interpretation operator.

The grouping in the delayed list is replaced by the delay of calculations operator, which differs from the delayed list grouping in a way that it returns only one value, the type of which is determined at compile time and can be anything. In a language with dynamic typing, the result was a parallel list. In the new model, issuing a swarm instead of a parallel list is also possible, but only if explicitly specified as a result of the delay. Disclosure of the delay occurs immediately after it becomes an argument of the interpretation operator. This allows in some cases to use this operator as a bracket expression that changes the priority of operations.

3 Static Data-Typing

Unlike the DFP programming language Pifagor, in which only basic data types are represented, the programming language Smile has a developed type system, due to the need to increase control at the compilation stage. The added basic data types largely repeat the types used in modern statically typed languages. However, besides this, types are offered that provide the ability to manipulate parallel lists, which leads to their definite effect on STMDFPC.

The following basic types are distinguished: integer, boolean, signal, functional, errors. These types are fundamental and are used not only in the processing of arbitrary data, but also in key language operators. Additional types, such as real numbers, characters, and others, are considered as extensions determined by the problem orientation, and can be included in various subject-oriented versions of the language. In general, it can be noted that issues related to the extension of the basic types are not critical at the level of the computational model.

Composite types include: array, structure, tuple, generalization, swarm, stream, functional type, reference type. These types are used to form derived abstractions defined by the programmer, and consist of both base and derived types. They basically replace the previously used concepts of a data list and a parallel list. However, they are descriptions and not operators, that allows them to form the corresponding data stores which use a single assignment principle. Array, structure, and tuple are specialized varieties of the DFMPC data list.

The array type is intended to describe data of the same type. In many ways, it is similar to using multidimensional arrays of traditional imperative programming languages. The array has fixed dimensions and lengths for each dimension. A description of this type at the programming language level is specified using the following syntax:

Array ::= TypeName «(» Dimension «)»
Dimension ::= Integer { «,» Integer }

Examples:

```
A << type int(100)
B << type bool(30, 40)
```

The structural type provides a grouping of data of different types by analogy with the structural types of various programming languages. The structure consists of fields, each of which has a name and type. The structure description has the following syntax:

Structure ::= «(» StructureField { «,» StructureField } «)»
StructureField ::= FieldName «@» TypeName
        | «[» FieldName { «,» FieldName } «]» «@» TypeName

Examples:

```
Triangle << type (a@int, b @ int, c @int)
Rectangle << type ([x,y]@int)
```

The tuple type differs from the structure in the absence of named fields. It is similar to an array, but may contain elements of various types. Access to the elements of the tuple is carried out by the field number. The following syntax is used to specify tuples:

Tuple ::= «(» TypeName { «,» TypeName } «)»

Examples:

```
C << type (int)
B << type (int, bool, signal)
```

The generic type is in many respects similar in organization and use to the generalizations used in imperative languages. Its main task is to describe variant data. There are various approaches to organizing generalizations, including methods that support polymorphism. The language uses generalizations that support the procedural parametric programming paradigm, which provides more flexible support for the evolutionary expansion of programs compared to other approaches [9]. The rules that define the syntax of generalizations are as follows:

Generalization ::= «{» GeneralizationField { «,» GeneralizationField } «}»

GeneralizationField ::= TypeName { «,» TypeName }
| TagName «@» TypeName
| «[» TagName { «,» TagName } «]» «@» TypeName

Examples:

```
Figure1 << type {Triangle, Rectangle}
Figure2 << type {trian@Triangle,
         rect@Rectangle,
         rhomb@Rectangle}
WeekDay << type{[Sun,Mon,Tue,Wen,Thu,Fri,Sat]@signal}
```

The swarm type is used to describe independent data on which large-scale parallel operations are possible. All swarm elements are of the same type, and the function that processes them can be simultaneously performed on each element. The result is also a swarm whose dimension is equal to the dimension of the swarm of arguments. The syntax rules defining this type are as follows:

Swarm ::= TypeName «[» Integer «]»

Example:

```
R << type int[100]
```

The data stream type is an alternative to an asynchronous list [10]. It is used to process data arriving sequentially and asynchronously at arbitrary intervals. The dimension of the incoming data is unknown, therefore, the completion of processing is possible only by the sign of the end of the stream. A stream is ready for processing if it has at least one element. The type of all stream elements is the same. The syntax rules that define the data stream are:

DataStream ::= TypeName «{» «}»

Example:

```
A << type int{}
```

The functional type allows us to specify the signature of the function by defining the type name, the type of the argument, and also the type of the result. The definition of a functional type differs from other languages only in that any function has only one argument and returns only one result. Syntax rules defining a description of a functional type are:

FunctionalType ::= func Argument «->» Result
Argument ::= TypeName | Tuple
Result ::= TypeName | Tuple

Examples:

```
F << type func int -> int
F2 << type func (bool, int, int) -> (int, bool)
```

The reference type provides support for pointers to various storages of a certain type, which allows us to transfer values between functions without copying them. Its main purpose is to provide additional type control during transfers. Syntax rules defining a description of a reference type are:

Reference ::= «&» TypeName
OpenArray ::= TypeName «(» «*» { «,» «*» } «)»

4 Function Descriptions and Static Type System

Unlike the Pifagor DFP programming language, an explicit specification of the argument and result types is used in function description, which provides additional control during compilation. These changes affect the function header, as defined by the following syntax description:

Function ::= func Argument «->» Result FunctionBody
Argument ::= ArgumentName «@» (TypeName | Tuple) | Structure
Result ::= TypeName | Tuple | Structure

Examples:

```
Factorial << func n@int -> int {...}
TrianPerimeter << func ([a,b,c]@int) -> int {...}
Sum << func t@(int, int) -> int {t:+ >> return}
```

5 Specifics of Instrumental Support

Adding a static type system to the language leads to the modification of tools that support dataflow functional parallel programming [5]. The developed language provides parallelism representation at the level of elementary operations, in which each function describes only the informational graph of the algorithm without any control relationships. The translator converts the source text of the function into an intermediate representation, which is used to optimize existing dependencies according to various criteria, as well as to build on its basis a control flow graph that defines the execution order in accordance with the chosen calculation management strategy [11]. Transformation of the control flow graph and its optimization make it possible to obtain strategies that differ from the dataflow control by data readiness and take into account various restrictions inherent to real computing systems.

A general diagram showing the various uses of the proposed tools is shown in Fig. 1. Within the framework of the created environment, the following subsystems are distinguished:

- a translator from the dataflow functional parallel programming language into an intermediate representation called a reverse data flow graph (RDFG);
- control flow graph (CFG) generator, forming a graph for computing control;
- an event machine that provides the execution of dataflow functional parallel programs in automatic and debugging modes, using RDFG and CFG as a program;
- optimization tools for a reverse data flow graph;
- control flow graph optimization tools;
- formal verification tools for DFP programs;
- toolkit for converting DFP programs into programs for other PCS architectures.

The translator is focused on processing text files, each of which contains one of the language artifacts. For each function, a reverse data flow graph is generated in the computer's memory, which is stored in the function repository in text form. The reason for choosing a textual representation for describing the RDFG is because the formation of an internal representation in the memory of a computer system on its basis can be easily performed using simple broadcasting programs. In addition, the developer can easily read and analyze the translated functions, considering this form as an analogue of the assembler language. Unlike the RDFG language with dynamic typing, this graph contains additional type information for each node.

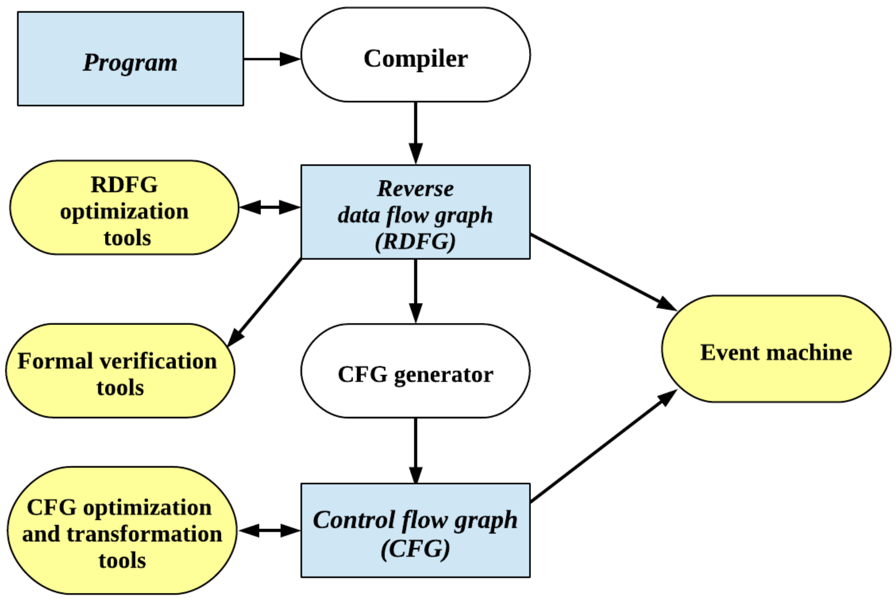
Program
Compiler
RDFG optimization tools
Reverse data flow graph (RDFG)
Formal verification tools
CFG generator
Event machine
CFG optimization and transformation tools
Control flow graph (CFG)

Fig. 1. The composition of the tools supporting dataflow functional parallel programming.

The reversible data flow graph generated by the translator allows us to build a control flow graph that determines the execution of the function. A special utility is designed for this, which generates a CFG that defines the management of RDFG vertices by data readiness. CFG is stored in the text form.

Testing and debugging of dataflow functional parallel programs at the current stage is carried out by a special interpreter (event machine), consisting of many event processors (EP), controlled by the event machine manager. Each of these processors (Fig. 2) carries out processing of only one function, launched in a separate thread. The operations inside the function are currently being performed sequentially due to the change in the state of the vertices of the CFG, which initiate the calculations at the vertices of the RDFG.

The functioning of the EP is as follows: the initial signals that record the flow of various events in the system and are determined by the initial marking of the CG are loaded into a queue from which they are transmitted to the processor in accordance with the service discipline. In the simplest case, this may be a FIFO discipline. The control signal processor analyzes the incoming event and selects the node of the control graph indicated in it. Based on the analysis of the state of the CG node, it can refer to the top of the information graph associated with it for the code of the operation being performed. In the case when the data processing operation is to be performed, a call is made to the RDFG node processor, which carries out the required functional transformations and saves the intermediate results. After processing the data, the control node switches to a new state and, if necessary, generates a signal transmitted to the next node, which enters the control signal queue.

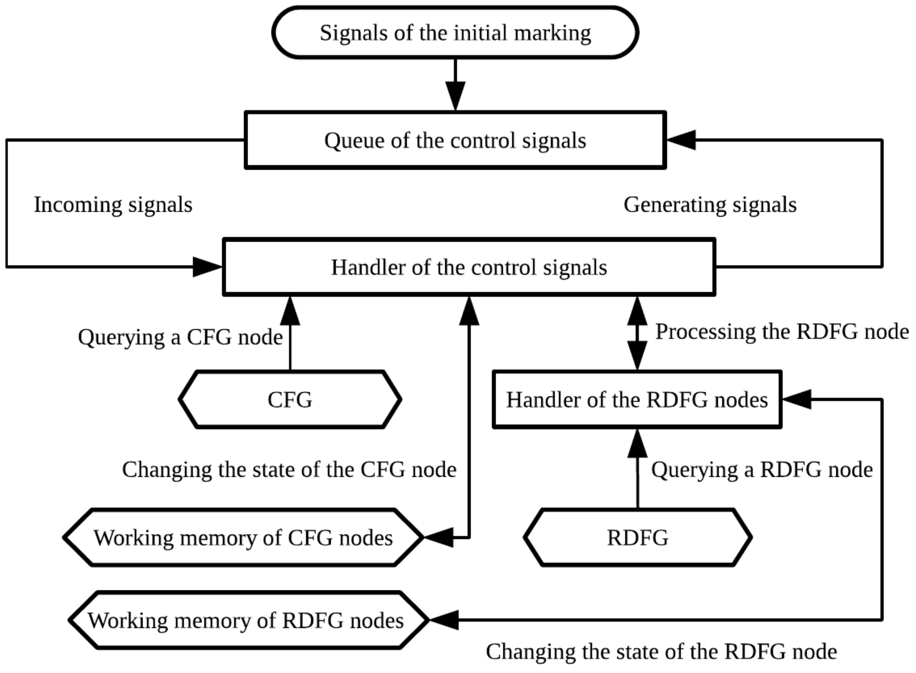
Signals of the initial marking
Queue of the control signals
Incoming signals
Generating signals
Handler of the control signals
Querying a CFG node
Processing the RDFG node
CFG
Handler of the RDFG nodes
Changing the state of the CFG node
Querying a RDFG node
Working memory of CFG nodes
RDFG
Working memory of RDFG nodes
Changing the state of the RDFG node

Fig. 2. The generalized structure of the event processor.

The main optimization methods developed at the present time involve the conversion of intermediate representations of dataflow functional parallel programs. They are aimed at changing the information and control graphs. The transformations are largely similar to the methods used to optimize the source code of programs and their intermediate representations in other programming languages, and are designed to solve similar problems. The specific of the dataflow functional model of parallel computing is own characteristics on the implementation of these methods. It is due to the algebra of equivalent transformations of the model implemented in the language: the information and control graphs can be changed independently of each other. In the course of optimization, it is necessary to ensure the consistency of RDFG and CFG, however, for many tasks, RDFG processing is sufficient. In such cases, the optimization of the control graph should be carried out after the transformation of the information graph and the construction of a new CG on its basis. It should be noted that the utilities currently being developed do not affect the distribution of real computing resources.

The presence of only information dependencies in the program and the absence of resource constraints make it easier for formal verification. The main tasks in this area of work are: study of the specifics of the application of formal methods of correctness proof and development of tools to simplify verification. The emphasis is on proving the correctness of the program using deductive analysis based on the Hoar calculus [12]. The Hoar triple is represented as an information graph of the program, to the input and output arcs of which formulas in the specification language (precondition and postcondition) are attached. The process of proving the correctness of the program consists in marking the arcs of the information graph with formulas in the language of specification, modification of the graph and its convolution. The result is several information graphs in which all arcs are marked. Each of the fully labeled graphs can be transformed into a formula in the language of logic. The identical truth of all the obtained formulas testifies to the correctness of the program. The methods developed for the Pifagor language [13] are also applicable to a language with static typing.

The proofing process is quite time-consuming, since it requires consideration of a large number of different versions of graphs and transformations. Therefore, the basic concepts of the architecture of a tool for supporting formal verification of programs in the DFP programming language have been developed [14]. The system receives at the input the information graph of the program and the precondition and postcondition formulas in the specification language. It finds unmarked arcs of the graph and helps with the selection of axioms and theorems necessary for their marking. The whole process of proof is presented in the form of a tree, each node of which is a partially labeled graph. The tree is completed when all its leaves contain fully marked up informational graphs of the program. After that, for each graph from the sheet, a formula is generated in the language of logic. If all formulas are identically true, then the program is correct.

6 Conclusion

The presence of static typing in the language of dataflow functional parallel programming provides more strict data control, which increases the reliability of developed programs. It also increases the possibility of more complete optimization and formal verification. In addition, the transformation of dataflow functional parallel programs into traditional parallel programming languages becomes easier and more effective, since most data types use almost single-valued mapping.

The reported study was funded by RFBR according to the research project No. 17-07-00288.

References

1. Levin, I.I., Dordopulo, A.I., Gudkov, V.A.: Programmirovaniye rekonfiguriruyemykh vychislitel'nykh uzlov na yazyke COLAMO. Uchebnoye posobiye. Izd-vo TTI YUFU. Taganrog (2011).

2. Dordopulo, A.I., Levin, I.I.: Resursonezavisimoye programmirovaniye gibridnykh rekonfiguriruyemykh vychislitel'nykh system. Superkomp'yuternyye dni v Rossii: Trudy mezhdunarodnoy konferentsii (25–26 sentyabrya 2017 g., g. Moskva), pp. 714–723. Izd-vo MGU, Moscow (2017).
3. Kasyanov, V.: Sisal 3.2: functional language for scientific parallel programming. Enterp. Inf. Syst. 2(7), pp. 227–236 (2013).
4. Legalov, A.I.: Funktsional'nyy yazyk dlya sozdaniya arkhitekturno-nezavisimykh parallel'nykh programm. Vychislitel'nyye tekhnologii 1(10), pp. 71–89 (2005).
5. Legalov, A.I., Vasilyev, V.S., Matkovskii, I.V., Ushakova, M.S.: A Toolkit for the Development of Data-Driven Functional Parallel Programmes. In: Parallel Computational Technologies. PCT 2018. Communications in Computer and Information Science, vol. 910, pp. 16–30. Springer, Cham (2018).
6. Vasilev, V.S., Legalov, A.I.: Loop-invariant Optimization in the Pifagor Language. Automatic Control and Computer Sciences 7(52), pp. 843–849 (2018).
7. Ushakova, M.S., Legalov, A.I.: Verification of Programs with Mutual Recursion in Pifagor Language. Automatic Control and Computer Sciences 7(52), pp. 850–866 (2018).
8. Udalova, J., Legalov, A., Sirotinina, N.: Metody otladki i verifikatsii funktsional'no-potokovykh parallel'nykh programm. Zhurnal Sibirskogo federal'nogo universiteta. Seriya «Tekhnika i tekhnologii» 2(4), pp. 213–224 (2011).
9. Legalov, A.I., Legalov, I.A., Matkovsky, I.V.: Instrumental support of the evolutionary expansion of programs using a incremental development. In: 20th Conf. Scientific Services and Internet, SSI 2018. Novorossiysk-Abrau. Russian Federation; 17–22 September 2018. In: CEUR Workshop Proceedings, vol. 2260, pp. 346–359 (2018).
10. Legalov, A.I., Redkin, A.V., Matkovsky, I.V.: Funktsional'no-potokovoye parallel'noye programmirovaniye pri asinkhronno postupayushchikh dannykh. In: Parallel'nyye vychislitel'nyye tekhnologii (PaVT'2009): Trudy mezhdunarodnoy nauchnoy konferentsii, Nizhniy Novgorod, 30 marta – 3 aprelya, pp. 573–578 (2009).
11. Legalov, A.I.: Ob upravlenii vychisleniyami v parallel'nykh sistemakh i yazykakh programmirovaniya. Nauchnyy vestnik NGTU. 3(18), pp. 63–72 (2004).
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM. 12(10), 576–585 (1969).
13. Kropacheva, M., Legalov, A.: Formal Verification of Programs in the Pifagor Language. In: Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. St. Petersburg, Russia. Lecture Notes in Computer Science 7979, pp. 80–89. Springer. (2013).
14. Ushakova, M.S., Legalov, A.I.: Automation of Formal Verification of Programs in the Pifagor Language. Modeling and Analysis of Information Systems 4(22), pp. 578–589 (2015).