DOI 10.31114/2078-7707-2019-2-40-45

Development of Capsule Programming Means for Recurrent Data-Flow Architecture

D.V. Khilko[1], Yu.A. Stepchenkov[2], Yu.I. Shikunov[3], G.A. Orlov[4]

The Institute of Informatics Problems, Federal Research Center "Computer Science and Control" of the Russian Academy of Sciences

[1]dhilko@yandex.ru, [2]ystepchenkov@ipiran.ru, [3]yishikunov@yandex.ru, [4]orlov.jaja@gmail.com

***Abstract* — This paper presents new results obtained in the course of work on the development of methods and tools for software programming and debugging of the multicore recurrent data-flow architecture (MRDA). At the current stage of development, the main goal is to automate the construction of a special programmer's tool – graph-capsules (GC), which visualizes the distribution of computing resources of the MRDA. To automate its creation, a component was developed to construct GC in numerical form, using the modelling results. The next step in the development of programming toolset is the creation of tools for graph and GC construction based on their symbolic form, which lays the foundation for the creation of the compilation tools in the future. This paper is dedicated to discussing the results of solving this problem.**

***Keywords* — data-flow architecture, data-flow graph, graph-capsule, capsule programming.**

I. Introduction

Most modern computing systems support parallel computing at various levels. The promising, but a not widespread direction of parallel computing systems research is the development of computing devices based on a data-flow architecture. One of the main advantages of such architectures is the "natural" support for parallelism due to the principle of computing based on data readiness. However, the functional prototype based on the data-flow architecture has not yet been created, due to a number of issues [1] – [3].

The Department of Architecture and circuitry basics of Innovative computing systems of The Institute of Informatics Problems in Russia have been developing multicore recurrent data-flow architecture (MRDA), which is approbated on the subject area of DSP. The main aspects of the MRDA are described in [4] – [6]. As [7] shows, principles incorporated into the architecture allow to partially or completely solve most of the problems inherent in data-flow architectures. Currently, the development of the MRDA is focused on three main areas: increasing its productivity, creating software development and debugging tools, and reducing data redundancy.

The theoretical and technical aspects of the architecture have been described in [7], [8], which cover the mechanisms and solutions that made it possible to achieve high performance of the MRDA prototype on the isolated word recognition problem. The problems of reducing data redundancy in data-flow architectures and the mechanisms for their solutions are discussed in detail in [9], [10]. As shown in [11] – [13], the key features of the architecture prevent the use of existing programming tools. Therefore, specialized software development and debugging tools are being created for the MRDA.

For the experimental approbation of the proposed architecture, its prototype has been developed: a recurrent signal processor (RSP), implemented in a hybrid two-level variant with a leading von Neumann processor at the controlling (upper) level (CU) and a number of data-flow processors at the lower level – recurrent operating unit (ROU) [14]. This prototype has been called the hybrid architecture of recurrent signal processor (HARSP).

Currently, the recurrent data-flow programming methodology [12], hardware-software modelling toolsets SIMPRA and SPRUT [15], [16] as well as the integrated development environment GAROS IDE have been created and deployed [13], [17]. The program in the MRDA is a sequence of self-sufficient data, which encode instructions for their processing. Such a presentation has been called a capsule, and the programming style itself – the capsular programming paradigm.

Further development of programming tools led to the need for refinement of both programming methodology as well as programming tools architecture. In [18], the initial results of programming methodology refinement are discussed, and a description is given of new development tools that allow a programmer to build a special tool — a graph capsule in numerical form. The experience of using this tool has shown the need for further development of programming methodology, as well as the creation of tools for constructing GC in symbolic form. This paper is covering the results of these developments.

II. Recurrent data-flow programming methodology

A. General description of methodology

Paper [12] covers the theoretical aspects of the MRDA programming process methodological support elements development. In particular, the paper defines the concept of a recurrent data-flow programming methodology, which

describes the software development process, starting from the mathematical description of the problem to the creation of capsules and data structures corresponding for them. The key stage of the methodology is the "Capsule Programming" stage, which describes the process of developing a capsule designed to solve a specific problem. Fig. 1 shows the overall structure of the methodology.

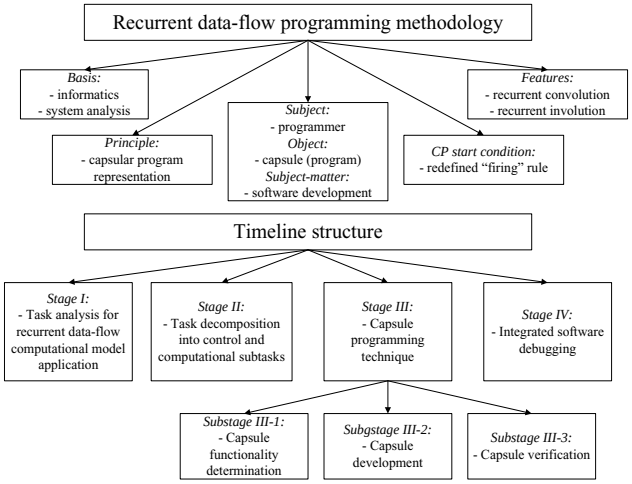

Fig. 1. Programming methodology structure

A capsule, like any other program, has a corresponding textual representation, which was called a symbolic capsule. To debug a symbolic capsule, both test and real data packets are used, based on which a numerical capsule is formed and modeled. Meanwhile, a legend of input data names is compiled.

The fundamental property of the MRDA ("recurrence") allows for the creation of programs with a dynamically generated computational process (CP) schema. On the one hand, this allows us to compress the original program, but on the other, it significantly complicates the development and debugging of such programs (capsules). Therefore, the new developer element was introduced into the methodology, a – "graph-capsule", which is a graphical interpretation of the process of unfolding the CP scheme encoded in a capsule.

The experience of programming and problem solving made it possible to organize an iterative capsule development process based on a data-flow graph, a GC, and modelling results. In this case, the GC is used as a means of verification. The use of GC as a primary debugging tool has proven its practical effectiveness. However, the manual construction of this developer tool is associated with significant difficulties. The problem of automating the construction of GC naturally arose. The solution of this scientific and technical problem has made it possible not only to shorten the development and debugging time of capsules by an order of magnitude but also to increase the visibility of capsule modelling results and to assess utilization degree of the architectures computing resources.

At the current stage of development of programming tools, it was possible to partially solve the problem of automating the construction of GC, but only in the numerical form based on the results of modelling a numerical capsule. The first operating experience of numerical graph-capsules is presented in [18]. This experience has shown that debugging and verification using a numerical graph capsule is associated with a number of difficulties that significantly reduce the effectiveness of its use by both the developers of behavioral models and the developers of the MRDA prototype. As an example, Fig. 2 shows a fragment of the data-flow graph (Fig. 1 from [18]) of the implementation of the Viterbi algorithm, and Fig. 3 and Fig. 4 (Fig. 4 and Fig. 5 from [18]) the corresponding numerical GC.

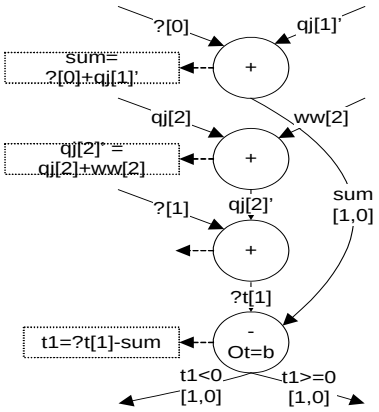

Fig. 2. Data-flow graph fragment (Fig. 1 from [18])

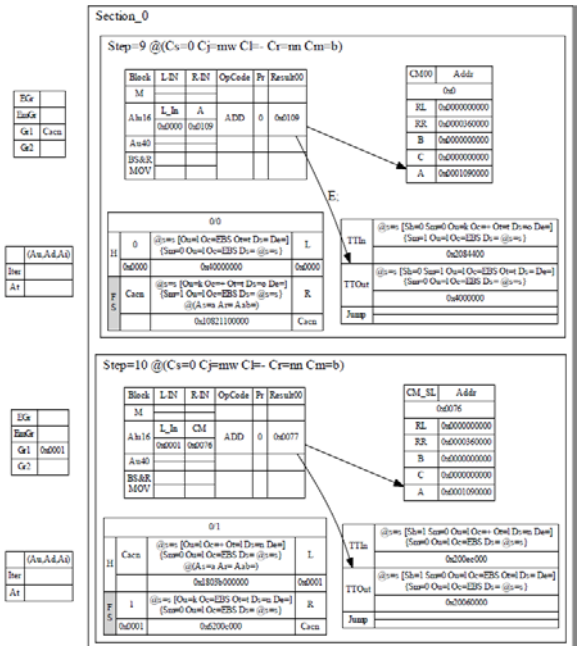

Fig. 3. Numerical graph-capsule, part 1 (Fig. 4 from [18])

B. Developments of methodology

One of the key substages (Fig. 1) of "capsular programming" is Substage III-2, which is called "Capsule Programming Technique" in [12]. In work [18], a refined method was proposed, according to which we construct**:**

- the expanded data-flow graph in symbolic form;
- folded symbolic data-flow graph;

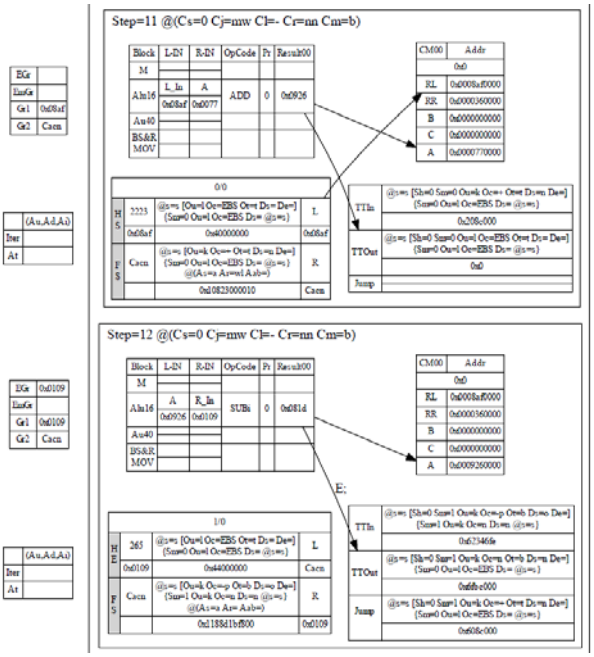

Fig. 4. Numerical graph-capsule, part 2 (Fig. 5 from [18])

- symbolic capsule;
- numerical capsule and numerical graph-capsule.

Next, we establish the identity between the symbolic data-flow graph and the numerical GC. One of the main problems at this stage is the comparison complexity of the symbolic data of the data-flow graph and the numerical data of the GC. In fact, such a comparison is only possible if the legend of the input data is stored, as well as with constant in-depth analysis of the operations performed at each step. In addition, Fig. 2 and Fig. 3 reveal a high degree of informational load of the GC, which greatly complicates its analysis. The need to constantly compare the observed numerical values with their semantic meaning significantly reduces the efficiency of the numerical GC. At the same time, the use of symbols is a more natural way of verification, since the results obtained are easily comparable with the original mathematical description of the problem according to sec. 1).

The need to introduce symbolic graph-capsules for verification became clear. The possibility of automated construction of both numerical and symbolic graph-capsules allowed us to logically separate the areas of their application. It is obvious that a detailed knowledge of the mathematical formulation of the problem being solved and the specifics of its capsule programming cannot be demanded from the hardware developer. But in the process of hardware debugging, the numerical graph-capsule turns out to be an extremely useful tool due to the visual interpretation of the CP in the hardware environment.

At the same time, at the first stages of software development, the programmer needs to be able to evaluate the logical correctness of the capsule being formed, for example: the correctness of operand pairing, the correctness of operand pairs distribution, the correctness of MRDA internal resources initialization, etc. Moreover, at this stage of the capsule development there is no need to solve the issues of accuracy and track the correctness of the numerical values of variables. Taking into account all the above, the symbolic graph-capsule becomes the most useful tool for developing a symbolic capsule.

Thus, the revised capsular programming technique will include the following steps:

1) Analysis of the mathematical description of the problem and the construction of the data-flow graph of the parallel algorithm;

1) Refinement of the data-flow graph in accordance with the limitations imposed by the specification of the MRDA prototype;

2) Building a dynamic data-flow graph by performing a recurrent convolution of the data-flow graph subgraphs;

3) Development of a fragment of a symbolic capsule that implements a part of a dynamic graph;

4) Creation and modelling of a fragment of a numerical capsule by means of GAROS IDE;

5) Construction of the symbolic and numerical GC based on the modelling results;

6) Verification of the program using the symbolic GC;

7) Verification of both the model and the prototype, as well as the program using a numerical graph-capsule;

8) A capsule is considered complete if the expanded data-flow graph is fully consistent with the corresponding graph-capsules, otherwise go to sec. 4).

To implement the updated methodology, it is necessary to introduce the means for constructing symbolic graph-capsules, and in the future, automatic verification tools into the programming toolset.

III. GRAPH-CAPSULES CONSTRUCTION AUTOMATION IN GAROS IDE

A. Description of GAROS IDE

The development of capsules for MRDA is fraught with many difficulties due to the fact that the capsular programming language is an “assembler” type language. In addition, data and instructions in a capsule are stored in a recurrently compressed form. Therefore, a specialized integrated development environment GAROS IDE is being developed, the functionality of which largely implements the main stages of the recurrent data-flow programming methodology. Fig. 5 shows the GAROS IDE architecture.

The “Decomposer” component that is intended for decomposition of the problem being solved (Stage II of the methodology in Fig. 1) has not yet been implemented. Third-party HLL tools (high-level languages) are a set of software tools for developing software that is executed at the controlling unit (CU). *The “Extractor”* component that is designed to extract parallelism based on the description of a task in a high-level language (substep III-1 in Fig. 1), is not yet implemented. *The “Graph”* component that is intended for manual or automated construction of data-flow graphs and GC, is partially implemented. It can be used to

build both symbolic and numeric GC (substage III-2 in Fig. 1). *Component "Capsule"* is designed to build capsules.

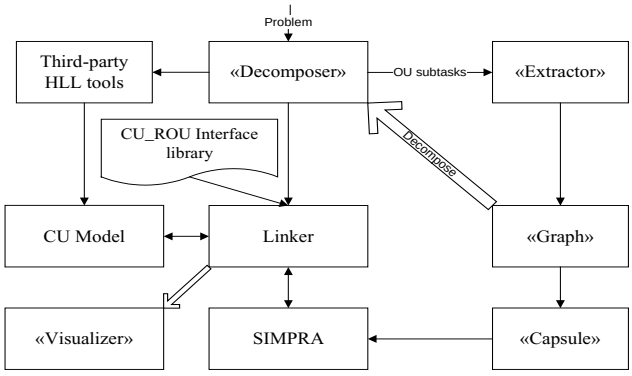

Fig. 5. General GAROS IDE architecture

The "CU Model" component is designed to interpret the control unit program. *The component "SIMPRA"* is the imitational modelling toolset, designed for modelling the behavior of the ROU. *The "Linker"* component is intended for organizing the interaction between the CU and the ROU modelling processes, taking into account the information stored in the auxiliary structures of these capsules. It performs the assembly and interpretation of the entire task as a whole. *The component "Visualizer"* is designed for displaying modelling results, as well as data-flow graphs and graph-capsules.

Thus, more than half of the development environment components are already implemented. Also, there is a constant modification of already developed components, in particular, the "Visualizer" component has been integrated into the "Graph" component. This paper discusses the results of extending the functionality of the GAROS IDE for building symbolic graph-capsules.

B. Symbolic graph-capsule construction technique

GAROS IDE modelling tools currently do not support symbolic modelling mode. Therefore, it is not possible to directly convert a symbolic data-flow graph to a symbolic graph-capsule. At the same time, the developed and successfully tested functionality of numerical graph-capsules construction can be effectively used for the construction of symbolic graph-capsules. Thus, the task can be formulated as follows: it is necessary to develop a method for converting a numerical graph-capsule into a symbolic one based on the expanded symbolic data-flow graph.

Said data-flow graph is constructed in accordance with the capsular programming methodology, and the following requirements are imposed on it:

• For each node, the concept of the context of the computational step number and the name of the parallel computational flow is introduced;

• A type is introduced for each node: common or composite (for superscalar modes);

• Each node describes the operation from the ROUs instruction set;

• Each arc is mapped to a datum and its symbolic name;

• For each datum a type is introduced: input, internal, intermediate, output;

• Multiple typing of arcs (data) is allowed;

• For each arc of "input" type, a correspondence is established between the L- or R-component of the pair.

Given these requirements, the method of converting a numerical graph-capsule into a symbolic one includes the following steps:

1) Conversion of the expanded data-flow graph in accordance with the requirements;

9) Construction of the corresponding symbolic capsule;

10) Construction and modelling of the corresponding numerical capsule;

11) Construction of the symbolic graph-capsule based on the modelling results and converted data-flow graph.

C. Graph-capsule construction tools implementation

To solve the automation of the symbolic graph-capsules construction in the GAROS IDE, the SIMPRA and Graph components were refined:

• the utility for constructing the expanded data-flow graph and the symbolic GC was added to the Graph;

• the modelling results history collection was expanded in the SIMPRA.

The paper [18] describes the first version of the numerical graph-capsule construction tool integrated into the "Graph" component and also presents the results of research of various graph construction and visualization libraries. The usage experience identified a number of flaws in the design and informativeness of the numerical graph-capsule: the lack of information about the internal type of the operands of the pair; low visibility of the use of internal resources of computing units, etc. In the updated version of the "Graph", these problems were also resolved, for both the symbolic and numerical graph-capsules.

The QuickGraph library was chosen as a tool for building and processing a data-flow graph. The GraphX library was used to visualize the data-flow graph, and the GraphViz utility was used for the symbolic and numerical graph-capsules. Using the QuickGraph library, the object structure of the expanded data-flow graph is built in the application memory, which then can be:

• transferred to the utility for constructing symbolic GC;

• saved to a file;

• visualized with GraphX.

The resulting data-flow graph, along with the modelling results, is used to build a symbolic GC. Due to the introduction of the context for each node of the data-flow graph, the

procedure for its comparison with the results of numerical modelling is greatly simplified. For each next context (step) during the construction of the symbolic GC the following actions are carried out:

• Symbolic names loading, assigned to internal resources in the previous step.

• A check is carried out on the availability of output data if they are available, then the symbolic name of the arc with the "output" type is substituted.

• For arcs of the "input" type with L and R markers, their symbolic names are assigned to the corresponding input data bus of the ROU computing unit.

• For arcs of the "internal" type, their symbolic names are assigned either directly to internal resources (if the arc is an output for the current node), or the name of the corresponding source resource is substituted (if the arc is an input for the current node), the resources are determined based on the analysis of the modelling history.

• For arcs of the "intermediate" type, the name is assigned to the result of intermediate calculations in the superscalar mode.

• As the modelling history is processed, previously asigned symbolic names are saved until the data is overwritten.

Due to the information provided in the nodes of the data-flow graph and typed arcs, it is possible to unambiguously map the data presented in the modelling history onto the name without requiring explicit linkage to the physical resources of the ROU.

As a demonstration in Fig. 6 and 7 we show the result of constructing a symbolic capsule graph corresponding to a fragment of a data-flow graph and a numerical graph-capsule presented in Fig. 2 - 4.

For greater clarity, we introduce the sequential numbering of the graph fragment nodes in Fig. 2 from 0 to 3. In node # 0, the performed operation is *sum= α[0] + qj[1]'*. On the symbolic graph-capsule (Fig. 6, Step = 9), we can easily identify a similar operation *sum=alpha [0] + qj[1]'*, while on a numerical graph capsule (Fig. 3, Step = 9) the same operation is performed, but its semantics is not obvious without storing and analyzing additional information about the CP.

Similarly, we can look at the nodes 1-3 in Fig. 2 and make sure that the symbolic graph-capsule allows us to easily verify and debug the logic of the capsule execution. In turn, the numerical graph-capsule helps us to achieve the required accuracy of calculations. Taking into account that both graph-capsule variants are built using the same modelling results, we can guarantee their one-to-one correspondence.

IV. Conclusion

The developed tools for constructing data-flow graphs and symbolic GC provide powerful and flexible capsule verification tools. The practice of their use has reduced the average development and debugging time of capsules by 2-3 times compared with the use of numerical GC.

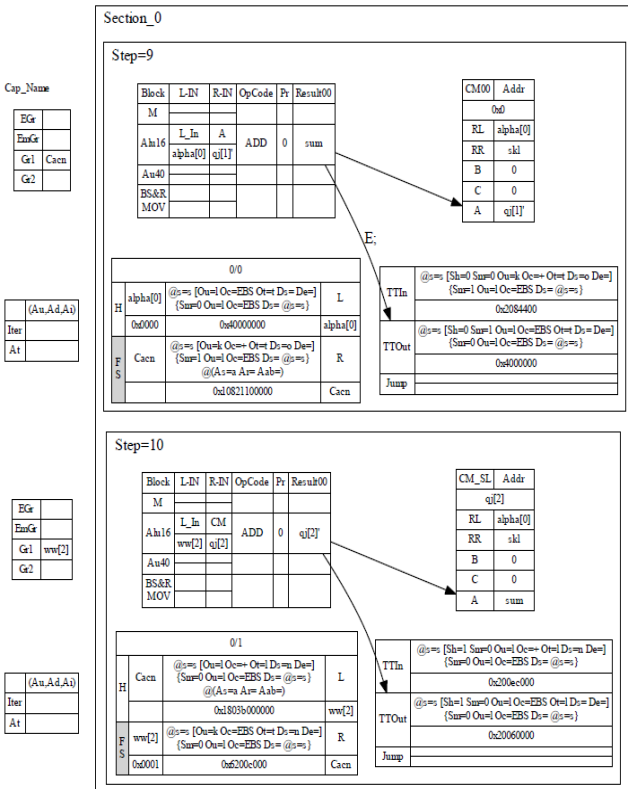

Fig. 6. Symbolic graph-capsule, part 1

The main scientific and practical result of this work is the development of methodological and software tools for developing and debugging of MRDA software:

• the introduction of the symbolic graph-capsule tool as the main means of verifying the program's execution;

• development of a numerical graph-capsule tool as the main tool for debugging a MRDA prototype.

In addition, the simultaneous use of an expanded data-flow graph and a symbolic GC made it possible to automate the process of searching for errors when designing a capsule. This is achieved by automating the analysis of the availability of ROU internal resources in the symbolic GC graphing utility.

Further development of the MRDA programming toolset is seen in solving the inverse problem, namely, in constructing a data-flow graph based on the symbolic GC. Creating such tools will automate the process of capsule verification at all stages of the capsular programming technique. The next step will be the development of the first versions of the tools for compiling and translating data-flow graphs directly into capsules.

Acknowledgements

In conclusion, we would like to express our gratitude to Morozov N.V., Dyachenko Yu.G. and Rozhdestvensky Yu.V. for the invaluable contribution to the development and debugging of capsules on the MRDA hardware model.

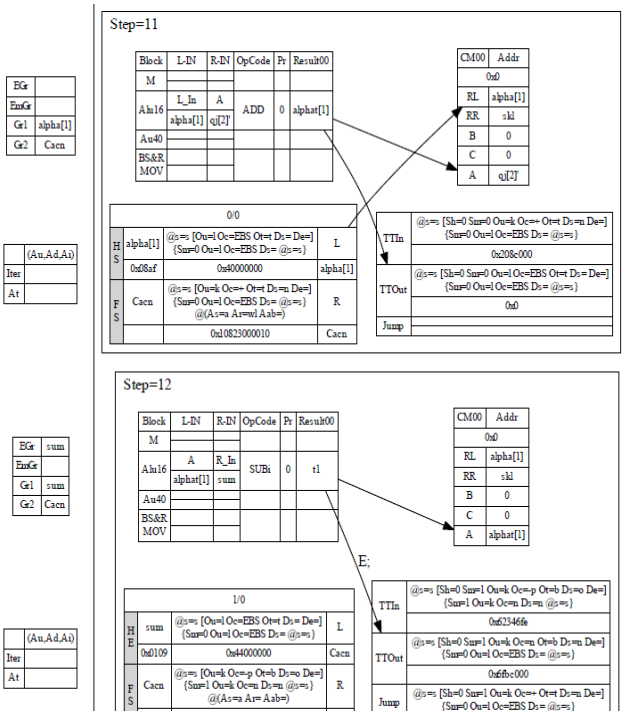

Fig. 7. Symbolic graph-capsule, part 2

SUPPORT

The research was carried out within the framework of state task No. 0063-2017-0011.

REFERENCES

[1] E.A. Lee and J.C. Bier. Architectures for Statically Scheduled Dataflow // Parallel Algorithms and Architectures for DSP Applications / edited by Magdy A. Bayoumi. Dordrecht. Kluwer Academic Publishers, 1991. P. 159-190.

[2] V.P. Srini. DFS-SuperMPx: Low-cost Parallel Processing System for Machine Vision and Image Processing // Proc. Third International Conference “Parallel Computing Technologies”, PaCT-95. St. Petersburg, 1995. Vol. 3. P. 356-369.

[3] S. Voigt, M. Baesler, T. Teufel. Dynamically reconfigurable dataflow architecture for high-performance digital signal processing // Journal of Systems Architecture, 2010, Vol. 56. Iss. 11, P. 561-576.

[4] Yu. Stepchenkov, V. Volchek, V. Petrukhin, A. Prokofyev. Hardware maintenance for digital processing of speech signals in the recurrent dataflow processor // Systems and means of informatics – TORUS PRESS, Moscow, 2010. P. 31-47 (in Russian).

[5] Yu. Shikunov, D. Khilko, Yu. Stepchenkov. Hardware and Software Modelling and Testing of Non-Conventional Data-Flow Architecture // Proceedings of the 2016 IEEE North West Russia Section Young Researchers in Electrical and Electronic Engineering Conference (ElConRusNW), 2016. P. 360-364.

[6] Yu. Stepchenkov, D. Khilko, Yu. Diachenko, Yu. Shikunov and D. Shikunov. Software and hardware testing of data-flow recurrent digital signal processor // Proceedings of IEEE East-West Design & Test Symposium (EWDTS’2016), Yerevan, October, 14 - 17, 2016. P. 168-171.

[7] D. Khilko, Yu. Stepchenkov, D. Shikunov, Yu. Shikunov. Recurrent data-flow architecture: technical aspects of implementa-tion and modeling results // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2016. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2017. Part II. P. 59-64.

[8] Yu. A. Stepchenkov, Yu. G. Diachenko, D. V. Khilko, V.S. Petrukhin. Recurrent data-flow architecture: features and realization problems // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2016. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2017. Part II. P. 52-58.

[9] Yu. Shikunov, Yu. Stepchenkov, D. Khilko, D. Shikunov. Data redundancy problems in data-flow computing and solutions implemented on the recurrent architecture // Proceedings of the 2017 IEEE Russia Section Young Researchers in Electrical and Electronic Engineering Conference (EIConRus), 2017 IEEE. P. 335 – 338.

[10] Yu. Shikunov, Yu. Stepchenkov, D. Khilko. Recurrent mechanism developments in the data-flow computer architecture // Proceedings of the 2018 IEEE Russia Section Young Researchers in Electrical and Electronic Engineering Conference (EIConRus), 2018 IEEE. P. 1413 – 1418.

[11] Khilko D.V. Programming tools of non-conventional multicore architecture and prospects for their development // Sbornik statej II regional'noj nauchno-prakticheskoj konferencii «Mnogoyadernye processory i parallel'noe programmirovanie» – Proceedings of II regional scientific-practical conference "Multi-core processors and parallel programming". Barnaul 2012. P. 62-70 (in Russian).

[12] Khilko D.V., Stepchenkov Yu.A. Theoretical Aspects of Recurrent Architecture Programming Methodology Development // Sistemy i sredstva informatiki, 2013, Vol. 23 no. 2, P. 133-156 (in Russian).

[13] Khilko D.V., Shikunov Yu.I. Software development environment creation for the recurrent data-flow computational model // Chetvertaya shkola molodyh uchenyh IPI RAN, 2013. Sbornik dokladov – Proceedings of fourth young scientists school IPI RAS. P. 65-77 (in Russian).

[14] Volchek V.N., Stepchenkov Yu.A., Petrukhin V.S., Prokofyev A.A., Zelenov R.A. Digital Signal Processor With Non-Conventional Recurrent Data-Flow Architecture // Problemi razrabotki perspektivnih mikro- i nanoelektronnih system (MES), Moscow, IPPM RAS, 2010. P. 412-417 (in Russian).

[15] Khilko D.V., Stepchenkov Yu.A. N, Shikunov Yu.I., Dyachenko Yu.G. The imitational modeling utilities of recurrent dataflow multicore architecture (SIMPRA). Version two. Certificate of state registration of computer programs № 2014610123 from 09.01.14.

[16] Khilko D.V., Stepchenkov Yu.A., Shikunov Yu.I., Shikunov D.I. The program complex for design and simulation of hybrid data-flow recurrent systems (SPRUT). Certificate of state registration of computer programs № 2017610828 from 18.01.17.

[17] Khilko D.V., Stepchenkov Yu.A., Shikunov Yu.I. The instrumental software development environment for hybrid architecture of recurrent signal processor (GAROS IDE). Certificate of state registration of computer program No. 2015614004 from 01.04.15.

[18] Yu. Shikunov, Yu. Stepchenkov, D. Khilko, G. Orlov. Graph-capsule construction toolset for data-flow computer architecture // Proceedings of the 2018 IEEE Russia Section Young Researchers in Electrical and Electronic Engineering Conference (EIConRus), 2018 IEEE. P. 1419 – 1423.