

Volume 21, Number 3
ISSN: 0361-7688

May-June 1995
CODEN: PCSODA

PROGRAMMING AND COMPUTER SOFTWARE

Official English Translation of *Programmirovaniye*

Editor-in-Chief
Viktor P. Ivannikov

Corresponding Member of the Russian Academy of Sciences
Professor, Director of the Systems Research Institute, Russian
Academy of Sciences

**A Journal of Original Papers and Reviews on
Theoretical and Applied Aspects of Computer Science**

Translated and Published by
MAMK HAYKA/INTERPERIODICA PUBLISHING

Distributed worldwide by PLENUM/CONSULTANTS BUREAU

Contents

Vol. 21, No. 3, 1995

Simultaneous English language translation of the journal is available from MAMK Hayica/Interperiodica Publishing (Russia).
Distributed worldwide by Plenum/Consultants Bureau. *Programming and Computer Software* ISSN 0361-7688.

Evaluation of Attributes in Attribute Grammars <i>V M. Kurochkin</i>	109
Strengthening of the Theorems on Polynomial Queries <i>A.B.Livchak</i>	113
Applications of Hypermedia Systems <i>J. Lennon and H. Maurer</i>	121
A Standard Syntax-Directed Editor for Hyperprogramming Systems <i>E. A. Zhogolevy E. A. Kuz'menkova, O. L Mailingova, and E. V Poprygaev</i>	135
Visual Syntax of the DRAKON Language <i>V D. Parondzhanov</i>	142
Languages and Interfaces for Facial Animation <i>N. Magnenat-Thalmann</i>	154

Visual Syntax of the DRAKON Language

V. D. Parondzhanov

Avtomatiki i Priborostroeniya NPO, ul. Vvedenskogo 1, Moscow, 117342 Russia

Received April 22, 1993

Abstract - A method for flowchart formalization and nonclassical structurization called DRAKON is suggested. The family of DRAKON visual programming languages is presented. The visual language syntax is described.

1. INTRODUCTION

In the development of on-board and ground-based software for the "Buran" orbiter, the programming languages PROL2, DIPOL, PSI-FORTRAN, LAKS, ASSEMBLER, and others were used. The first three were developed at the Keldysh Institute of Applied Mathematics, Russian Academy of Sciences. The experience gained from the application of these languages has resulted in the concept of the visual programming language DRAKON (the name is derived from the Russian abbreviation for Friendly Russian Algorithmic language That Provides Reliability). DRAKON is being developed at the Avtomatiki i Priborostroeniya NPO, in cooperation with the Keldysh Institute of Applied Mathematics, and is intended for the development of real-time software, as well as for educational purposes.

Improvement of program comprehensibility is among the most important requirements the DRAKON language should fulfill. This is connected with the fact that "the modern program of high quality should not only be efficient and reliable but also possess the most important properties of comprehensibility and supportability" [1, p. 17]. Further details on the great importance of comprehensibility can be found in [2-6].

It is well known that programming visualization is the most powerful means for improving program comprehensibility [7-9]. Flowcharts were used for this purpose [10, 11]. Recently, however, flowcharts have become subject to criticism [12 - 14]. Their opponents say that flowcharts are useless for structured programming [8, pp. 165 - 177; 15], cannot be formalized, and thus "cannot be used for direct input into a machine" [16], occupy many pages while "only very restricted material can be written in their cells" [17], "introduce additional inconvenience in education and decrease efficiency of comprehension," and in addition, are not equally acceptable for all. They are preferred only by "individuals with a dominant right hemisphere, those oriented toward visual information, intuitive ones, and those capable of pattern recognition; however, they are rejected by individuals with a dominant left hemi-

sphere who are oriented toward verbal information and inclined toward deductive reasoning" [18], etc.

Until 1980, flowcharts were "the most widely used means" [14], while today, they "are no longer regarded as necessary" [17], and "their popularity has decreased" [15]. In spite of a few attempts to adjust them for modern needs (SDL language, Cados project [19], etc.), flowcharts are obviously not part of the rapidly developing process of programming visualization, and their enormous potential capabilities are neglected.

The DRAKON language enables us to eliminate or at least significantly relax the disadvantages mentioned above. The term "DRAKON-chart" is used for the flowcharts designed according to the DRAKON rules.

2. DRAKON VISUAL LETTERS AND WORDS

Figure 1 shows the letters of the DRAKON alphabet. They are named "DRAKON-letters," or "icons." Letters are unified into the "DRAKON-words," a list of which is given in Fig. 2. In Fig. 1, icons 1 - 17 and 19 - 21 include a closed contour that must contain a text. The size of icon 22 can be enlarged, and it may contain not only text but also graphics and video, as well as controls for hypertext and multimedia. Icons 23 and 24 are ordinary textual comments (see [10], p. 10) that are placed to the left or right of the DRAKON-words.

A "skewer-block" is a part of a DRAKON-chart that has a single entry at the top and a single exit at the bottom, both on the same vertical. The latter is called the main vertical of the skewer-block. Icons 5 - 8, 10 - 17, 19, 21, and 22 in Fig. 1 and DRAKON-words 4 - 21 in Fig. 2 are examples of skewer-blocks.

A "terminator" is a part of a DRAKON-chart that has either a single exit at the bottom and no entry, or a single entry at the top and no exit. The icons 1 - 3 in Fig. 1 and DRAKON-words 1 and 2 in Fig. 2 are examples.

3. FAMILY OF THE DRAKON-LANGUAGES

The term DRAKON denotes a family of languages that contain DRAKON-1, DRAKON-2, DRAKON-BASIC, DRAKON-PASCAL, DRAKON-C, DRAKON-ASSEMBLER, etc. All the languages have

¹ Comprehensibility is "the program property of minimizing the intellectual effort required for its comprehension" [1].

	Drakon-letter	Name of the DRAKON-letter		Drakon-letter	Name of the DRAKON-letter
1		Title	15		Input
2		Cyclic start	16		Insertion
3		End	17		Pause
4		Formal parameters	18		Period
5		Headline	19		Start timer
6		Address	20		Timer-driven synchronizer
7		Action	21		Parallel process
8		Shelf	22		Comment
9		Question	23		Right comment
10		Choice	24		Left comment
11		Case	25		Loop arrow
12		Begin of FOR loop	26		Silhouette arrow
13		End of FOR loop			
14		Output			

Fig. 1. Visual alphabet of the DRAKON language

the same visual syntax, which is the visual standard of DRAKON, and differ only in textual syntax.²

DRAKON-1 is a visual pseudocode and, like a standard pseudocode or the PDL language [22], serves for the development of draft programs. DRAKON-2 is a lan-

guage for real time visual programming. The other languages (DRAKON-BASIC, DRAKON-PASCAL, etc.) are hybrids. A hybrid language, such as DRAKON-C, is obtained by unifying the visual syntax of DRAKON and the textual syntax of C according to certain rules.

Abstract drakon-charts, i.e., those whose icons are empty (not filled with text), are a language of "polyprogramming" in the sense of the program scheme theory [20]. They are thus a multi-language whose abstraction level is intermediate between the Martynyuk and standard schemes. The connection between abstract drakon-charts and program schemes is of a fundamental nature and leads to some interesting problems related to the fact that "the efficiency problem for translated programs grows into the automation problem for designing high quality programs" [21, p. 228].

4. EXAMPLES OF DIFFERENCES BETWEEN DRAKON-C AND C

Figure 3 shows seven examples of programs in C and equivalent programs in DRAKON-C. They facilitate understanding of the principles of design of a hybrid language.


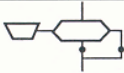

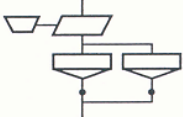

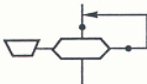
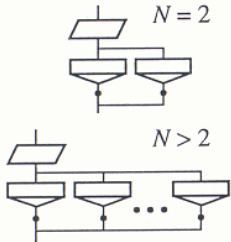
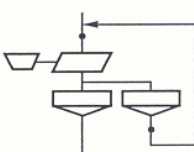

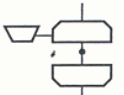
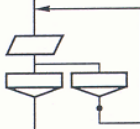
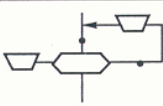

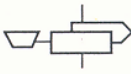
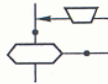
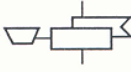
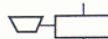

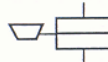

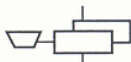
	DRAKON word	Name of the DRAKON word		DRAKON word	Name of the DRAKON word
1		Title with parameters	11		Timer-driven fork
2		Cyclic start with parameters	12		Timer-driven switch
3		Fork	13		Timer-driven ARROW loop
4		Switch (number of cases ≥ 2)	14		Timer-driven SWITCH loop
5		ARROW loop	15		Timer-driven FOR loop
6		SWITCH loop	16		Timer-driven WAIT loop
7		FOR loop	17		Timer-driven output
8		WAIT loop	18		Timer-driven input
9		Timer-driven action	19		Timer-driven insertion
10		Timer-driven shelf	20		Timer-driven start timer
			21		Timer-driven parallel process

Fig. 2. Visual words of the DRAKON language

To transform a C program into a DRAKON-C program, it is necessary to divide the C program code into two parts. The first part goes unaltered into a DRAKON-C program and is placed inside the DRAKON visual elements. The second part, which may be called "removable" or "parasite," becomes unnecessary and disappears, turning into graphical lines and keywords "yes" and "no."

Figure 3 shows that the list of C language parasite (removable) elements is rather impressive: it includes all the keywords in Examples 1-7, except for "default,"

all braces, brackets, parentheses, colons, and labels, the comments in examples 3 - 5 , and the semicolons in examples 2, 3, 7 and partially 6.

5. VISUAL SEMANTIC FRAMEWORK OF A PROGRAM

From the reader's point of view, any unknown non-trivial program is a problem that should be understood. To ease this problem, it is necessary to divide it into parts and reveal a semantic structure. The main difficulty is that none of the current existing programming

languages provide an effective aid to the reader for understanding the visual semantic framework of a program instantly. The DRAKON language has special means for providing a solution to the problem.

The top part of a DRAKON-chart is called a "cap" (see Fig. 4). This includes the algorithm header (the Title icon) and several Headline (branch name) icons (see Fig. 1). The purpose of the *cap* is to help the reader obtain an instant (in a few seconds) answer to the three following questions:

- (1) What is the name of the problem?
- (2) How many parts does it consist of?
- (3) What is the name of each part?

Here are the answers for Fig. 4.

What is the name of the problem? Bus journey. How many parts does it consist of? Four. What is the name of each part? (1) Search for a bus. (2) Waiting for a bus. (3) Entering the bus. (4) Bus trip.

To help the programmer formalize the semantic partition of a problem (program being designed), DRAKON provides a compound visual operator "branch" that has no analogs in the known languages. The division of the problem into *N* parts is performed by partitioning the program into *N* branches.

The branch has a single entry and one or more exits. An entry is represented as an Headline icon (see Fig. 1) containing a branch identifier. The visual operator "Headline" implements no actions; it serves only to declare the branch name, i.e., the name of the semantic part. Execution of a DRAKON program starts from the branch to which the Title icon points (see Fig. 1 and 4). The exit from the branch is represented as an Address icon containing the name of the next branch to be executed. The visual operator "Address" is the same as *goto*, but it transfers control only to the start of the chosen branch.

To clarify the main point, let us describe the DRAKON program in Fig. 4 with the aid of a usual textual pseudocode.

It is clear from Fig. 5 that some changes have been introduced into the traditional pseudocode to facilitate the description of branches. For instance, two new textual operators have appeared:

BRANCH (branch identifier)

ADDRESS (branch identifier)

The BRANCH operator of the textual pseudocode declares the branch name (written in visual pseudocode inside the Headline icon). The ADDRESS operator of the textual pseudocode transfers control to the textual operator BRANCH whose name is written in the operand field of the ADDRESS operator. Comparison of the two pseudocodes (see Fig. 4 and 5) shows that the visual one is certainly more comprehensible than its textual counterpart.

C language operators	Examples of programs in C	Examples of programs in DRAKON-C
① if-else	<pre>if (a >= 0) { x = f1; y = f2; } else { x = r1; y = r2; }</pre>	
② if-elseif-else	<pre>if (x < b) x = b; elseif (x < c) x = c; else x = d;</pre>	
③ switch, case, break, default	<pre>switch <n> { case 1: x = a; break; case 2: x = b; break; default: x = c; } /* switch */</pre>	
④ while	<pre>while (n++ < 50) { x = z + n; w = z - n; } /* while */</pre>	
⑤ do-while	<pre>do { y = p - a; z = p + a; } while (a-- > b); /* do while */</pre>	
⑥ for	<pre>for (n = 1; n < 20; n++) y = y + n;</pre>	

Fig. 3. Examples of programs in C and DRAKON-C

Another advantage is that graphics eliminate parasite elements. In the textual pseudocode, the following keywords turn out to be parasite: ALGORITHM, BRANCH, ADDRESS, END BRANCH, IF, ELSE, END IF, WAIT LOOP, END WAIT, COMMENT, GOTO, as well as labels.

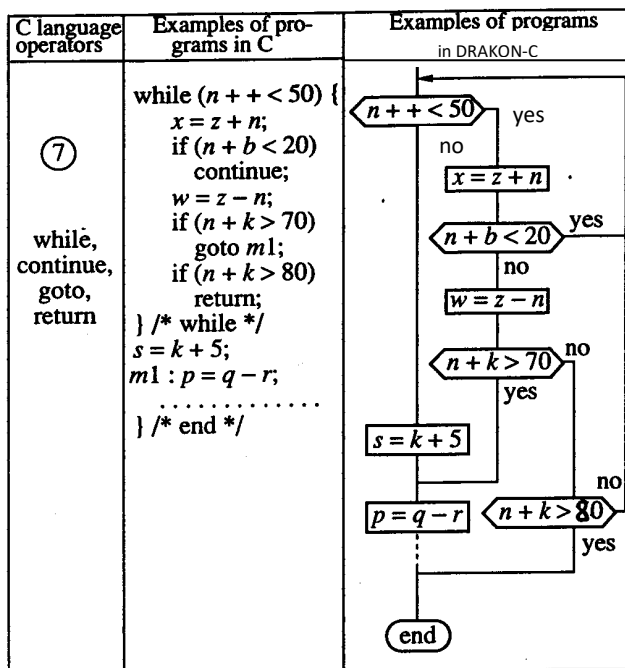


Fig. 3. (Contd.)

Unlike the textual pseudocode, DRAKON provides the reader with an efficient three stage method for comprehending actions of the unknown or forgotten program. At the first stage, in analyzing the *cap* the reader can comprehend the visual semantic structure of the program and its partition into significant semantic parts, i.e., branches. At the second stage, a deeper analysis of each branch is performed. At the third one, the interaction of branches is considered.

6. SILHOUETTE AND PRIMITIVE

A DRAKON-chart with branches is called "silhouette," and one without branches is termed "primitive." The silhouette in Fig. 4 can be represented as a primitive (see Fig. 6). Formally speaking, a primitive is a serial connection of skewer-blocks and two terminators: Title and End. An exit from the Title (starting terminator) and an entry into the End are always on the same vertical, which is called the main vertical (skewer) of a primitive. The main verticals of skewer-blocks also lie on this line (see Fig. 6).

The use of a primitive is recommended if a DRAKON-chart is quite simple and contains about 5-15 icons. Otherwise, a silhouette is preferable, since it improves program readability. Comparing Fig. 4 and Fig. 6 it is easy to see that a silhouette (Fig. 4) is more comprehensible than a primitive (Fig. 6), because a silhouette has a *cap* allowing you to understand the problem structure instantly. Moreover, large structural parts of the program (branches) are clearly

distinguished; they are visually and spatially separate, making a stable, recognizable, predictable, and integrated image. In a primitive, the structural parts are mixed together and not divided, which impairs readability and analysis of complex programs. However, a primitive is preferable for very simple programs that contain not more than 5-15 icons.

7. THE MAIN ROUTE

Consider the following problem. Given the complex labyrinth that connects the beginning and end of a complex program, it is necessary to select, in accordance with some criterion, a single route that serves as a sort of "leading thread." It should be possible to visually compare any other route to this thread to avoid getting lost among entangled paths. This thread (let us call it the "main route") should be easily distinguishable. In other words, casting a casual glance at a DRAKON-chart, we should see the distinct reference points that allow us to find easily and accurately the main route, as well as the other routes ordered with respect to the main one.

The notion of the main route has two aspects: visual and logical. From the visual point of view, the main route of a primitive is its skewer. Similarly, the main route of a branch is its skewer, i.e., the vertical from the Headline of the branch to its Address icon, and if the branch has several exits — to the leftmost Address icon. The main route of a silhouette is a serial connection of the branches' main routes in order of their execution.

Let us consider the logical aspect. The exit from a fork or switch that maximizes the success should be placed on the main route. This can be achieved by swapping the words "yes" and "no" at forks and transposing the switch cases, as well as their adjoint block sequences. Each branch should be constructed according to the single rule: verticals are placed from left to right in order of decreasing success from the actions they describe. For example, in Fig. 5 the "entering a bus" branch has three verticals. The left vertical (main route) describes the greatest success, since you have a seat. The right vertical describes the least success, since you have left the bus and the trip is delayed. The middle vertical (placed above the block "Do you want to travel standing?") occupies the intermediate position, since either partial success or failure takes place depending on the answer given.

In cases where the notion of "success" is irrelevant, another sensible criterion should be chosen so that a shift from the main route to the right would always have some reasonable meaning. For example, in the process of solving mathematical problems, exits from the fork and switch cases should be arranged from left to right in order of increase of mathematical quantities that correspond to them.

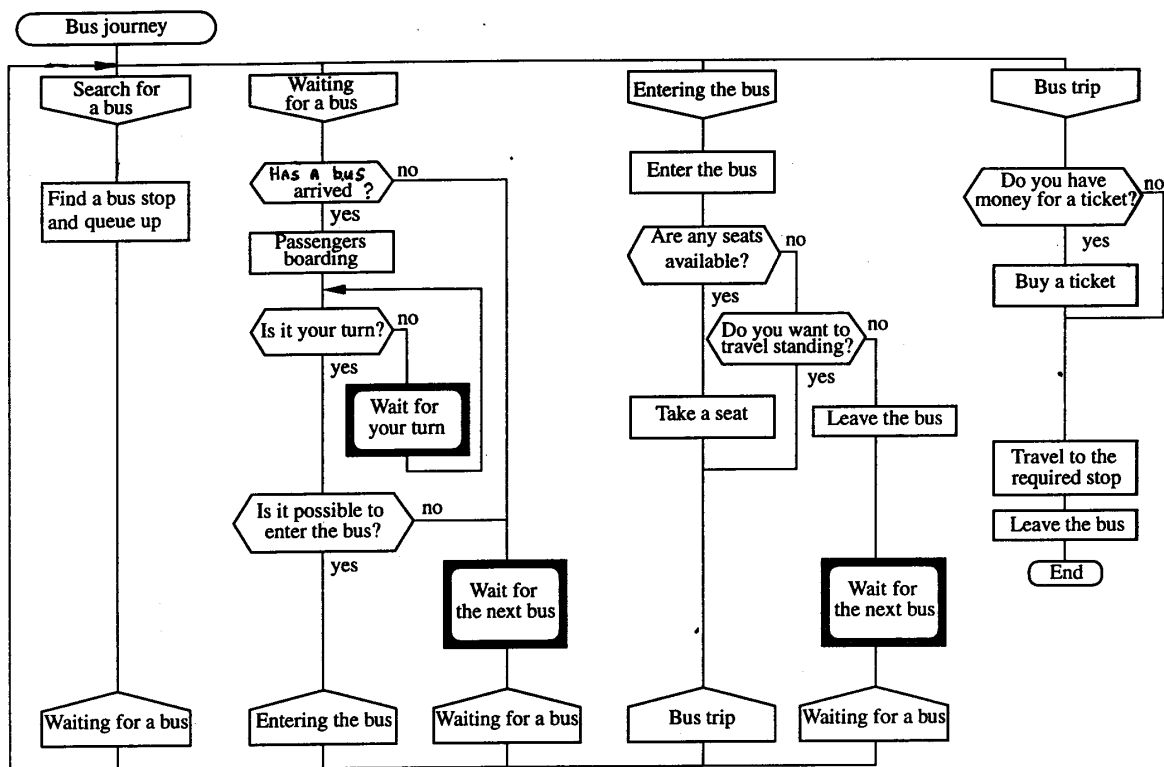


Fig. 4. The silhouette DRAKON-chart

```

ALGORITHM  Bus_journey
BRANCH  Search_for_a_bus
    EXECUTE  Find_a_bus_stop_and_queue_up
    ADDRESS  Waiting_for_a_bus
END BRANCH
BRANCH  Waiting_for_a_bus
    IF  Has_a_bus_arrived? = Yes
        EXECUTE  Passengers_boarding
        WAIT LOOP
        IF  Is_it_your_turn? = No
            COMMENT  Wait until it is your turn
        END WAIT
        IF  Is_it_possible_to_enter_the_bus? = Yes
            ADDRESS  Entering_the_bus
        M1:  ELSE
            COMMENT  Wait for the next bus
            ADDRESS  Waiting_for_a_bus
        END IF
    ELSE
        GOTO M1 .
    END IF
END BRANCH
BRANCH  Entering_the_bus
    EXECUTE  Enter_the_bus
    And so on

```

Fig. 5. The textual pseudocode corresponding to the graphic pseudocode in Fig. 4 (only two branches of four are described)

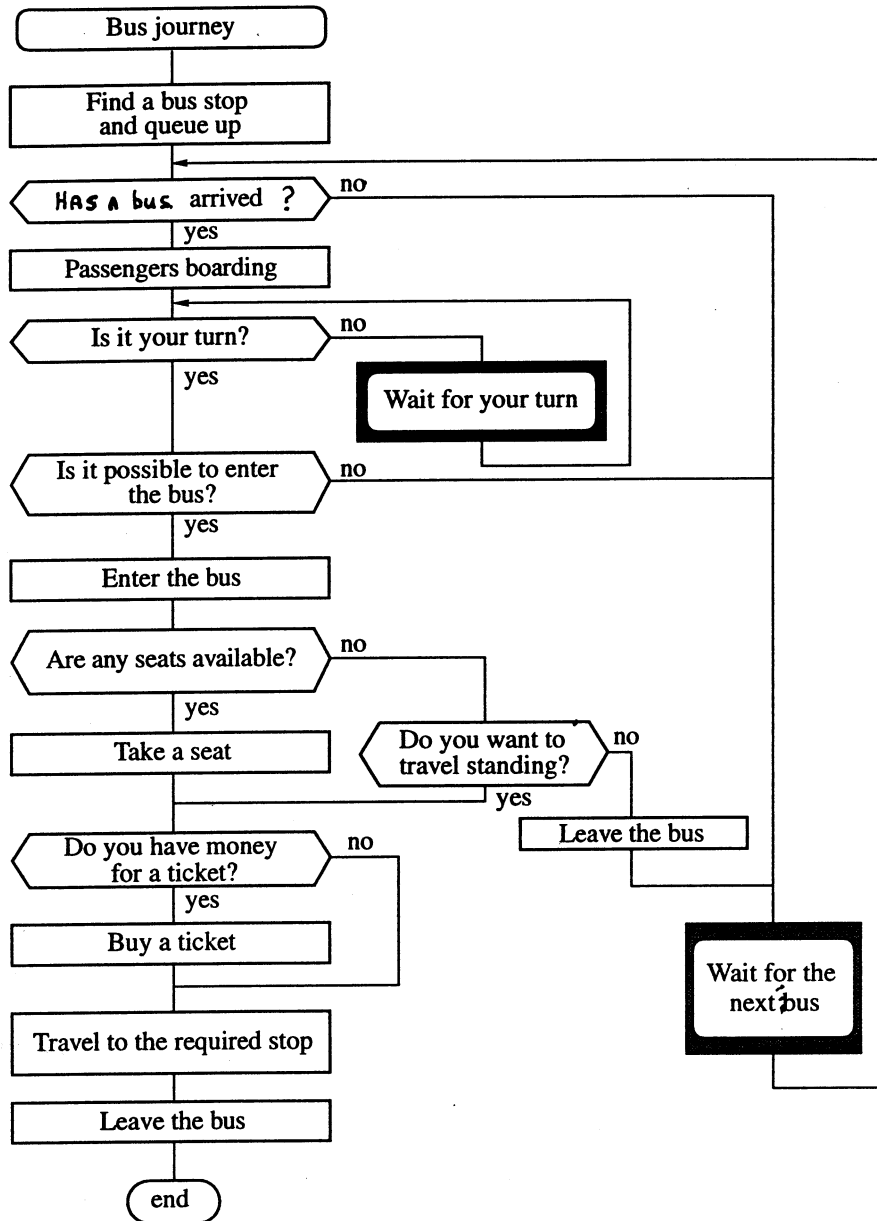


Fig. 6. The primitive drakon-chart equivalent to the silhouette in Fig. 4

Thus, DRAKON makes it possible for the reader to see the main route of a complex and branched algorithm at once. Moreover, it is easy to see that the shifts of verticals from the main route are not random, but meaningful and predictable, which improves understanding of the algorithm.

8. THE DRAKON LANGUAGE AND THE ASHCROFT-MANNA METHOD

When drawing standard flowcharts, two types of line crossings are allowed: an explicit crossing marked by a cross, and an implicit one made by so-called connectors. A connector "is used to break a line and continue it

elsewhere ... to avoid unnecessary crossings" (see [10, p. 10; 14]).

Both the described crossings are prohibited by the DRAKON rules, and the following question arises: is it possible to represent an arbitrary algorithm in the form of a DRAKON-chart?

Proposition 1. Any structured program can be written in DRAKON in two ways: as a primitive or as a silhouette.

Proposition 2. An arbitrary (unstructured) program cannot be represented as a primitive in certain cases; however, by means of some equivalent transformations,

admitting the introduction of additional variables (branch identifiers), it can always be represented as a silhouette.

Let us clarify the problem with examples. Figure 7 shows a forbidden drakon-chart, namely, a primitive with an unrecoverable (without introducing additional variable) crossing. Figure 8 shows a silhouette, which, as is easily seen, is equivalent to the primitive in Fig. 7, but does not contain illegal line crossing.³ Thus, the example in Fig. 7 and 8 confirms the validity of Proposition 2.

In addition, the primitive in Fig. 7 is an example of a unstructured program taken from [22, p. 140]. The silhouette in Fig. 8 is equivalent to the recursive structured program (see [22, p. 143]) obtained from the original program [22, p. 140] with the aid of the "advanced structuring method" (see [22, pp. 139 -146]) based on the method of introducing a state variable suggested by Aschcroft and Manna in [23] (see also [24]). The pseudorecursive construction named "silhouette" is equivalent to a combination of two constructions: the external loop and internal switch that are used in the Aschcroft-Manna method for obtaining a recursive structured program. Discussion in detail is beyond the scope of this article.

9. REAL TIME VISUAL OPERATORS

Figure 9 shows an example of a real-time program that has two inputs and one output. Calling up the OVERALL CHECK program, execution starts from the MOTOR CHECK branch. If the name PARTIAL CHECK is used, execution begins from the LOAD CHECK branch.

9.1. Operators "Pause," "Start timer," "Timer-driven synchronizer," and "Parallel process"

In Fig. 9, icon 19 "Start timer" contains the entry $T = 0$. This operator generates, sets to zero, and starts the virtual timer T . In the same branch, there is icon 20 (timer-driven synchronizer) with the entry $T = 1\text{min } 15\text{s}$ (1 minute 15 seconds). This means that if the conditions described in icon 9 are met, then the procedure TURN REACTOR ON will be called up only after the timer T counts 1 minute 15 seconds. The same branch contains icon 17 (Pause) with the entry 16s (16 seconds). This means that if the procedure TURN REACTOR ON is completed, then 16 seconds should lapse before calling up the procedure TURN LOAD ON.

The second branch contains two icons 21 that provide control for parallel processes. After completion of the procedure TURN LOAD ON, the parallel process PLASMA CONTROL is stopped, and the process ARC CONTROL started; the procedure STATION

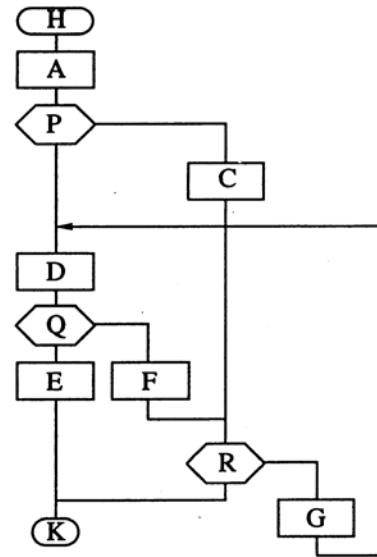


Fig. 7. An illegal drakon-chart: a primitive with line crossing

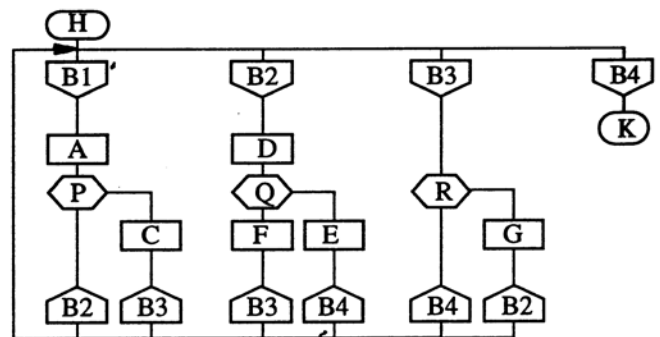


Fig. 8. A silhouette equivalent to a primitive in Fig. 7.

CHECK is then immediately invoked. The latter procedure works simultaneously and parallel with the process ARC CONTROL.

9.2. WAIT loop

Consider the following problem. Suppose that it is necessary to wait for three minutes until at least one of two events "Motor 1 norm" or "Motor 2 norm" occurs. When one of events occurs, the reactor should be turned on. If none of the events occur for three minutes, the station should be turned off.

Solving the problem in Fig. 9 requires two operators: "Start timer," which counts three minutes and WAIT loop. The latter contains the icon 18 (cyclic pause) and three icons 9 with entries Motor 1 norm, Motor 2 norm, and $T > 3\text{m}$. The last operator checks whether the timer value exceeds 3 minutes. If none of the events occur and the timer value does not exceed 3 minutes, control is transferred to the drakon-scheduler contained in real-time operating system (this is

³ Crossings "hidden" in icon 26 (see Fig. 1) are legal.

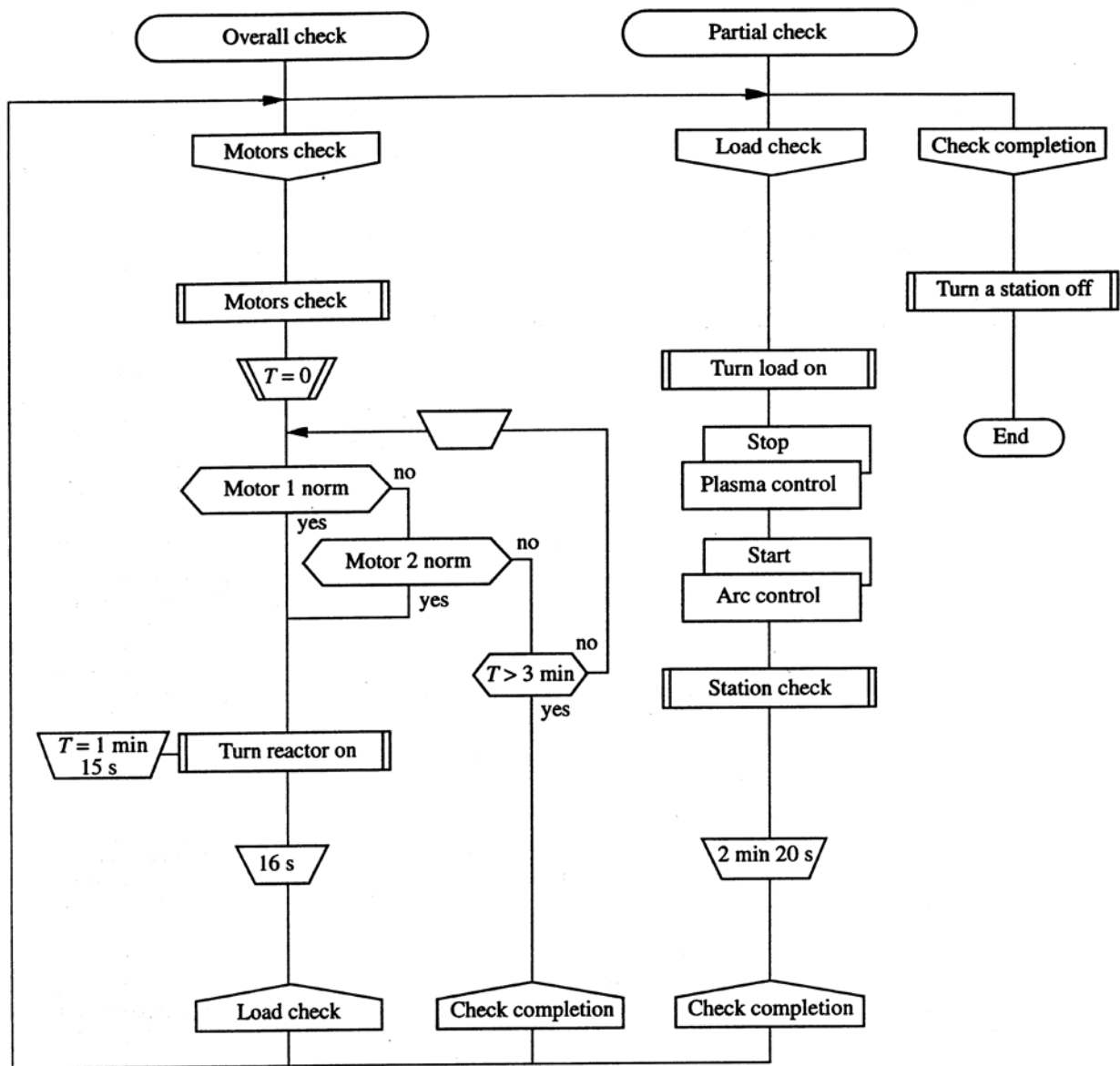


Fig. 9. An example of a real-time program.

shown in Fig. 9 by the empty Period icon. Then, the drakon-scheduler transfers control periodically (for example, every four seconds) to the start of the *WAIT* loop. It is clear from Fig. 9 that inquiries to the *WAIT* loop stop either at the moment when at least one of the expected events occurs, or when three minutes elapse and none of the events occur.

10. DESCRIPTION OF DRAKON VISUAL SYNTAX ВВОД

Definition 1. The valent point of the first kind is a point located on a connecting line for which the "insert" operation is defined. The set of valent points of the

first kind is defined by means of enumeration in Fig. 2 (see items 3-8 and 11-16). Note that in real drakon-charts, valent points are assumed but not shown.

Rule 2. The "insert" operation is performed as follows. A connecting line is broken at the valent point, and a skewer-block is inserted in this place (see Fig. 10).

All skewer-blocks are subdivided into three types: atoms, α -elements, and β -elements.

Definition 3. The atom is a drakon-letter or a drakon-word that is a skewer-block, except for icons 5, 6, 12, and 13 in Fig. 1.

Definition 4. The atom is called empty if it is equivalent to an empty operator.

An example of the empty atom is "fork" (item 3 in Fig. 2). To make a "fork" operator nonempty, it is necessary to place at least one nonempty operator in at least one valent point of the first kind in the fork.

Rule 5. Empty atoms are allowed for all stages of designing a drakon-chart, except for the last stage.

Definition 6. A sequence of atoms is a compound skewer-block, each skewer-element of which is an atom. This definition is equivalent to the formula

Sequence of atoms $::=$ atom {atom}

Rule 7. A set of α -elements is obtained from the set of sequences of atoms by multiple application of the "insert" operation according to rule 2, where "insert skewer-block" should be read as "insert an atom."

Nonempty atoms and α -elements may be called the structural elements of the DRAKON language, since they satisfy the postulates of classical structured programming [22, 24]. However, DRAKON has wider expressive capabilities. Along with α -elements, it enables us to design β -elements and γ -elements, which may be called metastructural. Formal metastructural elements help to overcome the known disadvantages of classical structured programming.

Definition 8. The valent point of the second kind is a point located at an entry or an exit of a skewer-block.

Definition 9. A liana is a line connecting any exit from a Question icon or Case icon, which is not on a skewer and is not a loop cycle, with another line.

Definition 10. Transplantation of a liana is a drakon-chart transformation performed as follows: the lower end of a liana is detached from its place and attached to any valent point of the first or second kind. The following conditions should be met: no other line is crossed, no other branch is touched, no new loop appears, and no second entry into a loop is created (Fig. 11).

Rule 11. The set of β -elements is obtained from a set of sequences of α -elements by multiple application of the "transplantation of liana" operation.

γ -elements have one entry and at least two exits. Therefore, unlike α - and β -elements, γ -element is not a skewer-block.

Rule 12. The "grounding of a liana" operation is performed as follows (see Fig. 12):

1. The end of a liana is detached from its place and is attached to any point on the lower bus (lower horizontal) of a silhouette to which it can be dragged without crossing other lines.

2. The Address icon is automatically inserted into the lower part of a liana by the "insert" operation.

Rule 13. The set of γ -elements is obtained from the set of sequences of α - and β -elements by multiple application of the "transplantation of a liana" and "grounding of a liana" operations.

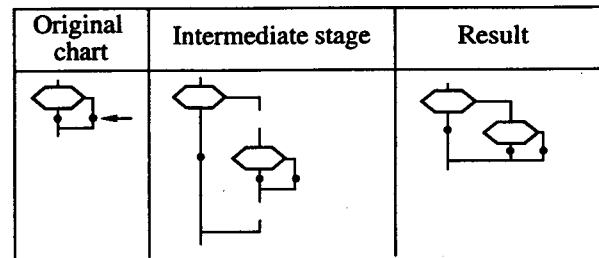


Fig. 10. An example of the "insert" operation.

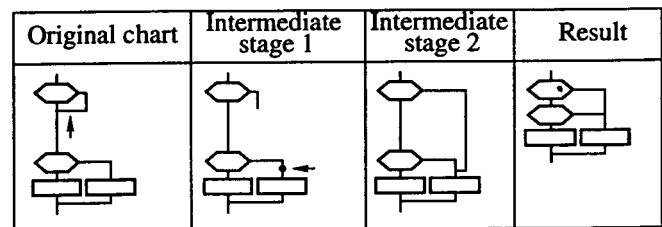


Fig. 11. An example of the "transplantation of a liana" operation.

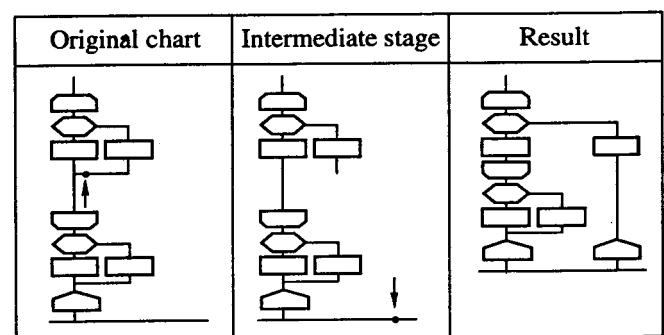


Fig. 12. An example of the "grounding of a liana" operation.

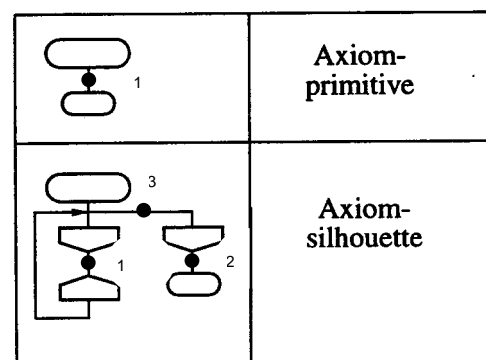


Fig. 13. Visual axioms.

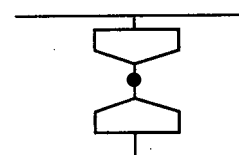


Fig. 14. Fragment.

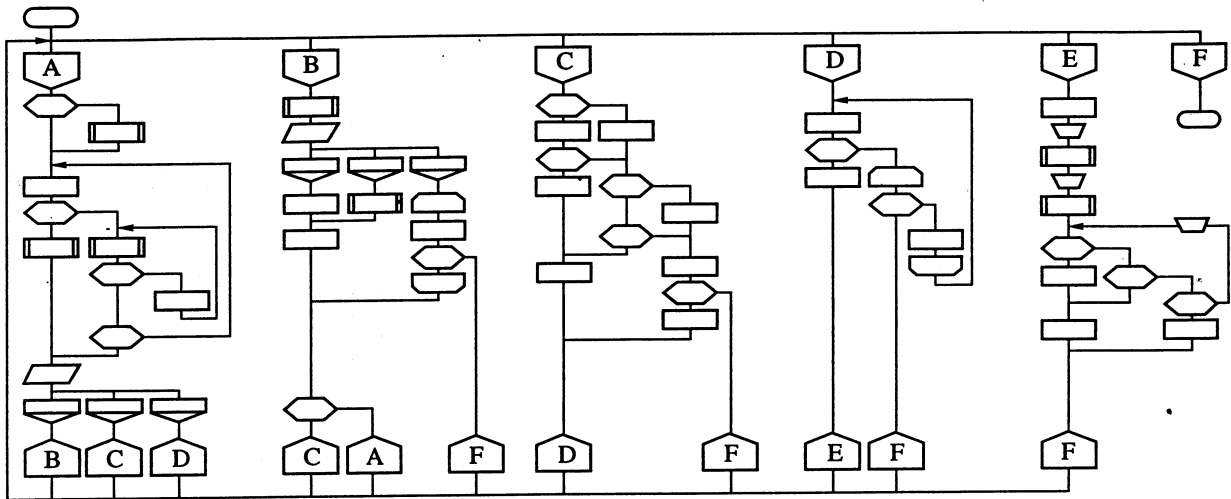


Fig. 15. A silhouette using α -, β -, and γ -elements.

Definition 14. The axiom-primitive and the axiom-silhouette are the prototypes shown in Fig. 13 that serve for the design of primitive and silhouette drakon-charts, respectively.

Definition 15. The points marked by digits 1, 2, 3 in Fig. 13 are called the axiom valent points of types 1, 2 and 3, respectively.

Rule 16. The "insert a skewer-block" operation is performed as follows: according to rule 2, a skewer-block is inserted into the axiom valent point of type 1 or into the drakon-chart valent point of types 1 or 2.

Rule 17. The "selection of an initial terminator" operation is performed as follows: icon 1 (see Fig. 1) in an axiom may be replaced by terminators shown by item 2 in Fig. 1 and items 1 and 2 in Fig. 2.

Rule 18. The "deletion of the end of a primitive" operation is performed as follows: in the "primitive" drakon-chart, the End icon and the entering vertical appendix are removed. This is necessary for describing an infinite parallel process.

Rule 19. Any correctly designed "primitive" drakon-chart is a result of transforming an axiom-primitive by the finite number of the following operations: insert a skewer-block, transplantation of a liana, selection of an initial terminator, and deletion of the End icon of a primitive.

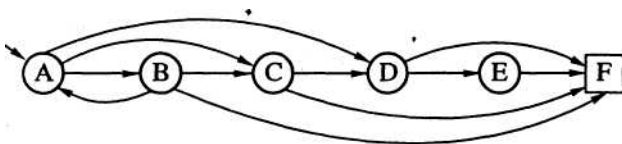


Fig. 16. A deterministic finite automaton corresponding to a silhouette in Fig. 15.

Definition 20. The fragment is a figure shown in Fig. 14. The fragment consists of an upper bus, a standard branch, and a lower bus.

Rule 21. The "Insert a fragment" operation is performed as follows: in an axiom-silhouette or a silhouette drakon-chart the upper bus is broken at the valent point of the third kind, and a fragment is inserted and attached to the upper and lower buses of a silhouette.

Rule 22. The operation "additional entry into a program" is performed as follows: an additional initial terminator is placed above the Headline icon (see the additional entry PARTIAL CHECK in Fig. 9).

Rule 23. The "deletion last of the branch" operation is performed as follows: the last branch and the entering vertical appendix of the upper line are deleted from a silhouette drakon-chart. This is used for describing an infinite parallel process.

Rule 24. Any correctly designed silhouette drakon-chart is a result of the transformation of an axiom-silhouette by a finite number of the following operations: Insert a fragment, insertion of a skewer-block, transplantation of a liana, grounding of a liana, selection of an initial terminator, additional entry into a program, and deletion of the last branch.

Example. Figure 15 shows a drakon-chart containing α -, β -, and γ -elements. The silhouette in Fig. 15 can be interpreted as a deterministic finite automaton [25], which is shown in Fig. 16 (an input alphabet and a transfer function of an automaton are not shown).⁴

⁴ The DRAGON language is described in detail in book by Parondzhanov V.D. "Learn to Draw Clear Flowcharts: DRAGON as an Accessible Visual Language for Systemizing Knowledge," which will be published in 1995 by Radio i Svyaz Publishers.

ACKNOWLEDGMENTS

I am grateful to my colleagues V.A. Kryukov, L.K. Eysymont, V.V. Lutsikovich, K.B. Fedorov, V.V. Baltrushaitis, G.N. Kostochkin, G.R. Kosenko, A.B. Aleshin, S.A. Kashinskii, A.I. Semenov, G.A. Gulenkov, S.A. Shcherbakov, V.G. Gora, L.D. Tyurina, and A.V. Kopylov for developing software for the "Buran" orbiter and for their ideas, advice, and friendly help.

REFERENCES

1. Sarkisyan, A.A., *Povyshenie Kachestva Programm s Pomoshch'yu Avtomatizirovannykh Metodov* (Improvement of Program Quality using Automated Methods), Moscow, 1991.
2. Robson, D.J., Bennet, K.H., Cornelius, B.J., and Munro, M., Approaches to Program Comprehension, *J. Syst. Software*, 1991, no. 14, pp. 79 - 84.
3. Parondzhanov, V.D., The Prospects for Information Technologies and an Increase of Productivity of Intellectual Work, *NTI, Series 1*, 1993, no. 5, pp. 8 -11.
4. Parondzhanov, V.D., The Crisis of Civilization and Unsolved Problems of Informatization, *NTI, Series 2*, 1993, no. 12, pp. 1 - 9.
5. Parondzhanov, V.D., What Fundamental Idea Will Win in the XXI Century? *Tekhnologiya Programirovaniya 90-kh Godov: Mezhdunarodnaya Vystavka-Yarmarka* (Software Engineering of the 90's: International Conference & Fair), Kiev, 1991, pp. 37 - 39.
6. Parondzhanov, V.D. The Unexpected Lessons of Cosmonautics of the XX Century: The New Role of the Human Factor, and the Cognitive Revolution in Information Technologies, *Tr. I Mezhdunar. Aviakosm. Konf. "Problemy Osvoeniya Kosmosa"* (Proc. 1st Int. Aerospace Conf. "Problems of Conquering Space"), Moscow, 1994, vol. 2 (Winged Space Systems).
7. Shu, N.C., *Visual Programming*, New York: Van Nostrand Reinhold, 1988.
8. Martin, J. and McClure, C. *Diagramming Technique for Analysts and Programmers*, New York: Prentice-Hall, 1985.
9. Khlebtsevich, G.E. and Tsygankova, S.V. A Visual Programming Style: Notions and Capabilities, *Programirovaniye*, 1990, no. 4, pp. 68 - 79.
10. GOST (State Standard) 19.701-90: Schemes of Algorithms, Programs, Data, and Systems: Notation and Drawing Rules.
11. Panteleeva, Z.T., *Grafika Vychislitelnykh Protessov*. (Graphics of Computational Processes), Moscow, 1983.
12. Brooks, Ph. R., Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company Reading, 1975.
13. Kak Proektiruyutsya i Sozdayutsya Programmnye Komplekсы. Mificheskii Cheloveko-Mesyats: Ocherki po Sistemnomu Programirovaniyu. — Moscow, 1979.
14. Glass, R.L. *Software Reliability Guidebook*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1979.
15. *Rukovodstvo po Nadezhnomu Programirovaniyu*. — Moscow, 1982.
16. Peters, L.J., Methods for Software Linkage and Representation, *IEEE Proc*, 1980, vol. 68, no. 9, p. 60.
17. Tolkovyi Slovar' po Vychislitelnykh Sistemam (The Explanatory Dictionary of Computer Systems), Moscow, 1991.
18. Verbitskii, I., Meet the R-technology, *NTR: Problemy i Resheniya*, 1987, no. 13, p. 5.
19. Fox, J.M. *Software and its Development*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1982.
20. *Programmnoye Obespechenie i Ego Razrabotka* — Moscow, 1985.
21. Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Inc. 1980.
22. *Psikhologiya Programirovaniya: Chelovecheskie Faktory v Vychislitelnykh i Informatsionnykh Sistemakh*. — Moscow, 1984.
23. Smith, K., Programmers' Work Productivity Increase under the Cados Project, *Electronics*, 1984, vol. 57, no. 23.
24. Kotov, V.E. and Sabelfeld, V.K. *Teoriya Skhem Programm* (Theory of Program Schemes), Moscow, 1991.
25. Kasyanov, V.N. *Optimiziruyushchie Preobrazovaniya Programm* (Optimizing Program Transformations), Moscow, 1988.
26. Linger, R.C., Mills, H.D., and Witt, B.I. *Structured Programming: Theory and Practice*. Addison-Wesley Publishing Company Reading. 1979.
27. *Teoriya i Praktika Strukturnogo Programirovaniya*. — Moscow, 1982.
28. Aschcroft, E. and Manna, Z., The Translation of "Goto" Programs into "While" Programs, *Proc. IFIP Congr.*, 1971.
29. Yourdon, E. *Techniques of Program Structure and Design*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975.
30. *Strukturnoe Proektirovanie i Konstruirovaniye Programm*. — Moscow, 1979.
31. Rayword-Smith, V.J. *A First Course in Formal Languages Theory*. Blackwell Scientific Publication, Oxford, London. 1983.
32. *Teoriya FormVnykh Yazykov: Vvodniy Kurs*. — Moscow, 1988.