Journal of Computer and Systems Sciences International, Vol. 39, No. 6, 2000, pp. 899–916.
Translated from Izvestiya Akademii Nauk. Teoriya i Sistemy Upravleniya, No. 6, 2000, pp. 63–81.

DISCRETE SYSTEMS

Software Automation Design: Switch-Technique: Algorithmization and Programming of Problems of Logical Control

A. A. Shalyto

Avrora, St. Petersburg, Russia

Received June 21, 1998

Abstract—This paper discusses foundations of a technique for algorithmization and programming of problems of logical control. The technique provides an increase in "safety" of software and can be called a state-technique or, more precisely, an automaton-technique. The corresponding field of programming is called automaton programming.

INTRODUCTION

This paper presents a technique for algorithmization and programming of problems of logical control called switch-technique. The urgency of its development is dictated, first, by the necessity of unambiguous and complete mutual understanding for the Customer, Technologist (Designer), Developer, Programmer, Operator (User), and Inspector, and, second, by the advisability of developing a unified approach to formal and a desirably isomorphic construction of "well-understandable" algorithms and programs for different types of controlling computers and programming languages able to solve problems of the class considered.

This issue is also urgent for the other classes of problems. For instance, in [1] E. Dijkstra writes:

> "On the one hand, I knew that programs could have a compelling and deep logical beauty. On the other hand, I was forced to admit that most programs are presented in a way fit for mechanical execution. Even if of no beauty at all exists, totally unfit for human appreciation. A second reason for dissatisfaction was that algorithms are often published in the form of finished products, while the majority of the considerations that had played their role during the design process and should justify the eventual shape of the finished program were often hardly mentioned."

The progress in solving this problem for tasks of logical control is particularly important in connection with the great responsibility of their solution for many control objects, e.g., for nuclear or chemical reactors. One of the prerequisites of this progress is the advanced mathematical technique of the theory of automata.

In the context of the technique developed, it is proposed to use two levels of languages, namely, languages of algorithmization (ALs) or specification (communication languages) and programming ones (implementation languages). Languages of these classes may both coincide (if a translator from an AL is available) and differ from each other.

For example, in a hardware implementation of logical control systems on the basis of switching circuits, as a language of algorithmization, functional diagrams (FDs) were used, and, as an implementation language, switching circuits (SCs) were employed. However, bad "readability" of both FDs and SCs led to the need for an intermediate language, namely, the functionally schematic diagrams. These diagrams reflect in the switching form only the control algorithm and do not contain other information typical for schematic diagrams (e.g., the designation of connectors, damping resistors, monitoring devices, and so on). At the same time, although these diagrams are notably convenient for representation of memoryless automata, they are difficult to read for automata with memory since the diagrams of this sort usually rather implement the dynamics of jumps and state changes of a synthesized automaton, than reflect it in their structure.

In software implementation on the basis of the Selma-2 hardware produced by ABB Stromberg (Finland) [2], for both an algorithmization language and a programming language, functional circuits are used. For the Autolog programmable logical controllers (PLCs) produced by FF-Automation (Finland) [3], the ALPro language of instructions are used as a programming language, while the algorithmization language is not specified. It is also not specified for many other types of PLCs (such as Melsec of Mitsubishi Electric (Japan) [4]) whose programming languages are the language of instructions and ladder diagrams language, and which provide a language of switching circuits supplemented by a great number of computing operations.

Presently, with algorithmization languages in systems of logical control, the ladder and functional diagrams, as well as flow diagrams of algorithms, also called graph diagrams of algorithms (AGDs) or diagrams of algorithms are most frequently used. For pro-

gramming languages, three types of languages are used depending on the types of controlling computers, namely, high-level algorithmic languages (for example, C, Pascal, PL/M, and Forth), low-level algorithmic languages (assembler and languages of instructions), and special-purpose languages (for instance, functional and ladder diagrams).

Below, we substantiate the advisability of the use of controlling graphs (jump graphs) as an AL for description of algorithms with memory. We also propose a unified methodological approach to the implementation of these algorithms on the basis of programming languages of different types. This makes it possible to produce for such languages the same algorithmic description independent of the type of controlling computer employed.

In logical control, we can use conventional (classical) methods of formalization of both control procedures and descriptions of control objects, which distinguishes this class of problems, e.g., from the case control [5].

For important technological objects, we can use systems of logical and case control in combination.

1. CLASSICAL LANGUAGES OF LOGICAL CONTROL

1.1. Boolean Functions, Truth Tables, and Decision Tables

In logical control systems, Boolean functions (BFUs) and systems of BFUs given in the form of truth tables (TTs) for completely defined functions and decision tables (DTs) for incompletely defined functions are conventionally used. Note that TTs that describe automata with memory are called coded jump tables or coded transition and output tables.

Application of TTs is restricted to problems of a low dimension and the use of DTs is limited basically by memoryless automata (combination diagrams). Table representation of automata with memory is not easy-to-interpret.

1.2. Boolean Formulas and Other Analytical Forms of Representation of Logical Control Algorithms

The analytical form of representation of Boolean functions is provided by Boolean formulas (BFs) and systems of BFs (SBFs), which make it possible to describe combination diagrams as well as automata with memory of a large dimension. SBFs can be isomorphically represented by ladder or functional diagrams. Sometimes, it is also possible to use other forms of analytical representation of Boolean functions, for example, threshold [6], spectral [7], or arithmetic [8, 9].

The main limitation on the use of SBFs for automata with memory is the poor obviousness of these functions.

1.3. Functional Diagrams

The advantages of FDs in their use as an algorithmization language are conventionality and the uniqueness of description including parallel processes. Among their disadvantages are: (1) the use in most cases of binary internal variables stored in triggers, whereas they are implemented by hardware tools that allow us to process many-valued variables; (2) the impossibility to show the values of output and input variables in the diagram; (3) a complexity of their reading (understanding) to completely represent the sequential process implemented by them; (4) the problem of choice of tests for their complete check; (5) the complexity of guaranteed modifications.

Note that reading of functional diagrams is replaced with computations for individual circuits to determine values of the output variables for different sets of input variables. In this situation, even for a comparatively small number of inputs, it is very difficult to find out basing on the functional diagram what factors affect on one or another transition in this diagram and to completely represent the behavior of even a comparatively small fragment of the circuit when triggers and feedbacks in this fragment are applied. For instance, for a circuit with three interconnected triggers, it is very difficult to directly find out (without calculations) basing on this circuit how many states it implements, because three triggers can code from three to eight states.

It should be noted that the use of input–output relations (which provide completeness of checking for memoryless circuits) as tests does not solve the problem of analyzing all functionalities for circuits with memory implemented by application of feedbacks (automatic interlocking) and/or triggers since, in this case, it is also necessary to check the validity of the order of changes of variables. However, despite this, it is these relations that are applied presently in developing methods for checking the operation of the majority of logical control systems. This does not guarantee their proper checking because these methods do not allow us to analyze all jumps on the set of states of the circuit. Moreover, these transitions are not known since the construction of circuits of this class of systems is usually carried out heuristically, without the use of the concept *state*.

Functional diagrams in their application as a programming language have all advantages of declarative languages of functional programming [10]. This book states that their main advantage is their functionality (referential transparency), i.e., each expression defines a single value, and all its references are equivalent to the value itself. The fact that one may refer to an expression from another program unit has no effect on a value of this expression. This property defines a difference between mathematical functions and the functions that can be written in procedural programming languages (such as Pascal), which enable functions to refer to the global data and use "destroying" assign-

ment. This assignment may lead to side effects, e.g., to a change of the value of a function in its call even if the values of its arguments have not been changed. Such a function is difficult to use since to determine the value obtained in its computation, it is necessary to consider the current values of the global data. In its turn, this requires us to analyze the *prehistory* of computations to assess what generates this value at every instant.

Under specific conditions (renotation), in systems of Boolean formulas employed in the construction of functional diagrams, even for automata with memory, it is possible to provide another advantage of declarative languages, namely, the independence of results from the order of calculation of formulas.

1.4. Temporary Diagrams and Cyclograms

The merit of these forms of representation of algorithms consisting in the representation of the dynamics of processes and their disadvantage is the practical impossibility to provide information of all feasible values of output (and, especially, internal ones) variables even for problems of a relatively low dimension. Therefore, in practice, these diagrams are usually constructed for the description of the "basic" mode and the entire algorithm is specified only in the program, which, for this reason, in many respects, is informally constructed basing on the diagrams.

1.5. Graph Diagrams of Algorithms

One of the merits of AGDs in their use as an algorithmization language for systems of logical control is a possibility of representation by them in an explicit form of the sequence of events (defined by values of input variables) and reactions to their appearance (presented in the form of the values of the output variables including those calculated in parallel). The availability of binary values of variables written in an explicit form in operator nodes dramatically simplify the understanding of AGDs (in this case, they are called automaton AGDs) compared to functional diagrams.

AGDs have the following disadvantages:

(1) In the literature, two kinds of automaton AGDs are used. For graph diagrams of the first kind it is implicitly supposed that the input of input variables is performed at each conditional node and the output of values of output variables is carried out at each operator node [11]. For graph-diagrams of the second kind it is supposed that the input of input variables is performed at the beginning of the body of the graph diagram and the output of values of output variables is carried out at the end of this body. The use of the operators *input* and *output* in the explicit form in such AGDs makes it possible to differ the versions of graph diagrams specified above.

(2) There are no requirements on what the graph diagram must represent. The alternatives are as follows: a control algorithm (AGDs with internal feedbacks, but without internal variables); an algorithm of execution of the control algorithm (AGDs without internal feedbacks); an algorithm that takes into account the properties of control structures of the programming language employed (AGD that is linearized and structured, probably, in a special way); an algorithm of execution of a program (AGD, in which components of the processor, for example, accumulator, are mentioned).

(3) There are no requirements on their organization (except structuring (structurization)) that guarantee the ease of "reading."

(4) The necessity (in the general case) of their multiple transformations to provide the possibility of solving several problems by one control computer (uncycling) and the account for the properties of control structures of the programming language (for instance, linearization and structuring).

(5) The availability of internal (intermediate) variables absent in the "verbal algorithm" of logical control (these variables that dramatically complicate the reading of AGDs by other Specialists different from the Developer, and, especially, by the Customer).

(6) Common use of a great number of bit internal variables, each of which not only has to be set, but compulsorily dropped as well. These variables characterize only individual components of states of an automaton, and its entire states are usually not described. The use of these variables is natural in hardware implementation of algorithms, but, in their implementation by programming languages that enable us to process many-valued variables the application of bit internal variables is inexpedient.

(7) The availability of flags and default internal and output variables at operator nodes. This hinders reading of AGD in view of the necessity to remember the prehistory, especially, in the cases where values of variables at these nodes change depending on the ways by which one may "arrive" at the node considered.

(8) Checking the values of only individual binary variables at conditional nodes. This leads to an awkwardness of AGDs.

(9) The link of operator nodes through conditional points. This hinders modifications, because modification of the conditions for transition between two operator nodes has influence on conditions of jump at other nodes of this type.

In the application of AGD, for most cases going from algorithmization to programming for complex problems of logical control there is a serious difficulty. This is explained by the fact that, usually, a process of algorithmization rarely, if ever, is completed properly, i.e., by the generation of an algorithm in a mathematical sense, which, by definition, must be uniquely executed by any Computer and ends only with a "pattern" called an algorithm. This pattern is to be supplemented to some extent in programming (for example, it may be necessary to structure an AGD or introduce uncondi-

tional jumps into an unstructured program). In that situation, either the Developer has to program himself or the Programmer has to know all specificities of technological process, or they together have to eliminate unavoidable errors of the conventional program design during tests.

Let us consider in more detail default values of input variables at operator nodes of automaton graph-diagrams.

Automaton AGDs of the first type are split into two classes. For the first class, at each operator node, the notation of only those output variables or their inversions that take at it only unit values is presented. The other output variables take zero values by default. For the second class of graph diagrams of this type, it is typical that each operator node, the notation of only those output variables or their inversions that take at the node unit and zero values, respectively, is presented. For every not specified variable, it is assumed that the node holds the previous value.

For automaton AGDs of the second type, at each operator node, the notation of those output variables, that take at this node fixed values, is indicated in the explicit form, and the renotation of those output variables that hold at this node the previous value may be also presented. It is assumed that each of the unspecified variables holds its previous value.

For automaton AGDs (except for the first class of the first type), there is also the possibility to store the previous values of all output variables on "wires", i.e., without the use of operator points in a corresponding circuit of a graph diagram.

The possibility to store the previous values of output variables at operator nodes or without the use of them hinders dramatically the understanding of graph diagrams.

The considered language is employed by the Opto firm (USA) for programming of PLC Mistic [12]. The use of an AGD in the form of Nassi–Shnenderman diagrams [13] does not eliminate practically the mentioned disadvantages.

1.6. Logical Diagrams of Algorithms

ALDs proposed by A.A. Lyapunov [14] are the string form of the notation of the linearized ALDs (LALDs) and are formed by letters (that correspond to conditional, unconditional, and operator nodes of LALDs) and are enumerated by arrows that indicate jumps executed if the conditions are not satisfied.

ALDs provide a compactness of a description, but they suffer from the lack of obviousness and are very difficult to read.

2. UNCONVENTIONAL LANGUAGES OF DESCRIPTION OF LOGICAL–CONTROL ALGORITHMS

2.1. SDL

The ideas of the theory of automata were employed in the Specification and Description Language (SDL) [15] developed by International Commission on Telephony and Telegraphy whose diagrams are similar to AGDs at the first glance, but differ from them by introducing states into these diagrams in an explicit form. However, SDL-diagrams are very cumbersome and correspond to only one class of automata, namely, Mealy automata [11], which results in serious disadvantages.

2.2. R-Charts

Another model based on Mealy automata is proposed by I.V. Vel'bitskii [16]. This model is called R-chart. An R-chart is oriented graph with weighted arcs depicted by vertical and horizontal lines and consisting of structures with only one input and output. Charts of this class have two types (one of them is special) of nodes and arcs along with certain types of connective lines. R-charts are formed by three types of connections: sequential, parallel, and nested.

This language makes it possible to reflect the structure of algorithms in a more compact form compared to AGDs. However, the application of the nonstandard notation and of only one type of automaton models restricts the use of these charts.

2.3. Petri Nets and Operation Graphs

For a description of complex processes including parallel ones, in 1962, A. Petri [17] proposed a graph model named after him. The model consists of nodes of two types, namely, positions and jumps connected by edges, and two nodes of the same type cannot be directly connected. To reflect the dynamics, a set of labels placed at positions is introduced. If all positions connected by entering arcs with a jump are labelled, then this jump comes into action and the labels jump into positions connected by outgoing arcs with the jump considered. For purposes of control, S.A. Yuditskii [18] proposed to employ only safety and living Petri nets (PNs). Positions of a safety PN cannot have more than one label. Living PNs have the fundamental possibility of operating at any jump. Let us call the specified class of PNs controlling Petri nets. Petri nets, in which each jump has only one entering and outgoing arc are called automaton Petri nets.

As a model for description of control algorithms, S.A. Yuditskii [18] proposed to employ operation graphs (OGs). They are controlling Petri nets in which positions and jumps are labelled by values of the output and input variables, respectively. For a description of

hierarchically constructed algorithms, he proposed to use systems of embedded OGs.

The advantage of operation graphs is a possibility of description (in an obvious graphic form including the form of one component) of complex control algorithms possessing parallelism, whereas their limitations and disadvantages are as follows:

(1) Parallel processes in the majority of cases must be synchronized, whereas, for many algorithms of logical control, this is not necessary.

(2) For OGs constructed on the basis of automaton Petri nets only a model of a Moore automaton [11] is employed, and it is impossible to employ other automaton models. This dramatically restricts the graphic possibilities of the operation graphs.

(3) For coding of positions, we may employ only "unit" codes. In this case, the number of internal variables (without taking into account their renotation) is equal to the number of positions including positions for operation graphs constructed on the basis of automaton Petri nets. Note that, for this class of graphs, all positions may be coded by a single many-valued variable if the approach proposed in this paper is applied.

(4) In the implementation, the system of embedded operation graphs is transformed into a single component, while, when using the proposed approach, the number of components in the description and implementation can be the same.

(5) It is recommended that one label the positions of operation graphs rather than those values of input variables that are changed in a corresponding position by all their values. Due to the use of defaults, in the general case, the complexities of description of the algorithm and its behavior are different. This hinders the reading of the algorithm and the analysis of all its functionalities.

2.4. GRAPHSET

This graphic language elaborated in the Space Research Center in Toulouse (France) is used now in parallel with the other languages [19] by different firms such as *Telemecanique* (France), Siemens (Germany), Allen Bradlay (USA), Toshiba (Japan), Omron (Japan). This algorithmization language, given a proper language translator, is a programming language as well.

GRAPHSET mainly differs from the language of operation graphs by the form of representation. It uses squares (instead of circles for the notation of positions) and rectangles (for the notation of values of output variables) not presented in the operation graphs. Therefore, all advantages and disadvantages of the language of the operation graphs are also typical for GRAPHSET. Translators from this language designed by different firms (at least by the firms specified above for their own control computers).

One of the advantages of GRAPHSET-diagrams consists in the standardization of their representation: diagrams are placed primarily in the downward direction. Simultaneously, this is their disadvantage, since, for integral (gestalt) human perception of "patterns," it is more expedient to use their planar representation (as in graphs of jumps), which, for the reason mentioned, makes it possible to represent algorithms more compactly.

GRAPHSET, despite the above mentioned disadvantages, is incorporated into the software of PLCs brought out by the world's leading firms in the automation field. For example, Siemens gives an opportunity to write programs for its PLCs using in the STEP-5 language (languages of instructions, ladder and functional diagrams) as well as in the S7-GRAPH language (GRAPHSET) [20, 21].

At the same time, experience suggests that having several programming languages for the same PLC, developers usually employ languages that are more conventional for systems of logical control such as ladder and functional diagrams. In many respects, this is because of the insufficient scientific and methodological support of the use of controlling graphs for the specification of algorithms. Moreover, in the documentation of many firms, it is usually proposed that one construct ladder and functional diagrams heuristically without preliminary description of algorithms by controlling graphs. The efficiency of such graphs compared, for instance, with ladder diagrams is demonstrated in rare cases (Omron) only by examples without the presentation of the method of formal transition from a controlling graph to a "diagram."

Note that, for different models of controllers of the same firm [22], it is proposed that one employ different languages (ladder diagrams for "lower" models and GRAPHSET for "higher" ones). At the same time a unified specification language is not used for all models.

The above-presented information is quite natural, since, in the field considered even the terminology has not been established yet. For example, the Sequential Function Chart (SFC) term simultaneously means jump graphs, AGDs (Opto), and GRAPHSET (Omron). In documenting *Telemecanique*, it is noted that GRAPHSET-diagrams are known as SFC and, as specified above, Siemens uses for the same purpose another term, namely, GRAPH.

Furthermore, the SFC term does not reflect the main characteristic property of the language considered, i.e., a possibility of representation in a single graph of processes that are parallel in states. From the theory of automata it is known that, in order for describing sequential processes, one may use a deterministic-automaton's jump graph, which gives no way of reflecting processes parallel in states.

2.5. *Problem-Oriented Languages Close to the Natural Ones*

There exist several task-oriented languages developed for logical control. They are intended for a formal linguistic description of algorithms of the class considered. These languages are also called [23] primary, since, in the opinion of the authors of the paper mentioned, they are primarily oriented toward the Customer and have the developed descriptive tools and constructions applied in the assignment of working conditions in a natural language.

The specified advantage of these languages implies a number of difficulties and disadvantages. The main of which are the following:

(1) They require that Participants of the design study the syntax and semantics of a new language (which is of limited circulation) with a sufficiently large number of constructions.

(2) They are constructed on the basis of rather a natural language (for instance, Russian) than of a mathematical one.

(3) They have poor obviousness of "texts" compared with graphs in the representation of structure, interaction, and dynamics of processes.

(4) They do not allow us to verify formally the completeness, consistency, and lack of generation, as well as to perform optimizing transformations.

(5) They require the development of a multilevel jump system that uses different types of languages such as core, automaton, and machine ones.

(6) They are oriented to a specific type of a core or automaton language and a specific type of automata.

(7) The complexity of verification and lack of tests for correctness of description.

(8) The complexity of correct modifications.

(9) Impossibility of their application if a translator for the control computer exploited is not available.

Among the languages of the class considered, the best theoretical justification is obtained by the following languages of logical control and their modifications: Forum [24], Uslovie [25], Upravlenie [26], and Yarus [27].

For example, the constructions of the Uslovie primary language are translated first into the core language of the operator diagrams of parallel algorithms with memory (this core language is a development of AGDs) and, then, they are translated into the automaton language, namely, the language of excitation and output functions.

In the use of the Upravlenie language, by analogy with labelling of an AGD for construction of automaton [11], binary labels considered as states are introduced in the source linguistic description (program code) of a program. Then, according to the labelled description, a jump graph (JG) of Mealy automaton is constructed. Problems of minimization of the number of states and parallel decomposition are solved on its basis [26].

The most approximate approach to that proposed in this paper is employed in the development of the Yarus language. Here, O.P. Kuznetsov proposed to use for description of operation of "items" an automaton model called a switch graph [28]. This model is a JG of a Mealy automaton with a default of fixed values of output variables. A possibility of reduction of the number of nodes of this graph compared to the conventional automaton models leads to the replacement of the "state" term with the "situation" term. It is worth noting that this simultaneously deteriorates the readability of the graph because of *the appearance of dependence on the "deep" prehistory*.

Contrary to the information presented above, for the technique described in this paper, it is primary and the formalized description is not a linguistic one, but automaton description of sequential processes with the help of transition graphs. The type of the JG, their number, method of coding, and interconnection are not fixed in the general case and depend on the problem solved. Clearness of such a description for the Customer is provided by the riser and simplicity of syntax of this language in the description of the statics and, above all, of the dynamics of the processes. The diagram of connections control automaton (CA)–control object (CO) (this diagram must define semantics of every external variable employed in a JG) has to be also developed. The application of a JG without flags and defaults [29] makes it possible to provide directly in the description of a process its correctness and use it as a test for verifying a formally written program. At the cost of elimination of flags and defaults, the modification of the program is simplified and the *dependence on the deep prehistory is eliminated* (further, the prehistory is the future that depends on the present, but does not depend on the past).

Clearness of algorithms and programs developed on their basis improves even more if the dependence of values of the output variables on the values of the input variables is also eliminated. This is achieved by the use of the JG of the Moore automaton whose values of the output variables depend only on the number of the automaton's state.

A jump graph of an automaton of this type is directly constructed or obtained from a transformation of the jump graph of an automaton of another type. For example, in terms of a JG of a Mealy automaton one may construct a JG of an automaton without an output transformer, which is a graph of accessible labelling of the source graph, which, in its turn, is transformed very simply into the JG of a Moore automaton.

We can assume that each node in the JG corresponds to a state of an automaton of the considered type, for which the graph is constructed, and a state of the memory of a computer provides software implementation of this graph.

However, since the behavior of an automaton is described most obviously by a JG, rather than a corresponding graph of accessible marking [30], the actual number of states of the automaton coincides with their number of this graph or of the equivalent to it, a JG of a "conventional" (without flags and defaults) Moore automaton.

Thus, if, in the general case, a JG can be considered as a coded (the number of nodes of the JG may be sufficiently less than the number of actual states of the automaton) and, therefore, compact description of the behavior of the automaton, then a JG of the classical Moore automaton in many-valued coding of its points nodes (at the same time, this JG is a graph of its accessible labelling) is the most clear. It is precisely a JG of this type that we propose to be employed as a basic specification language for problems of logical control.

In the technique developed, a transition to a linguistic description is executed in algorithmization, rather than just in programming and only on the basis of algorithmic languages. In this case, *all* structural specificities and properties of selected automaton model have to remain the same as possible in the program text. This is guaranteed only for a unique and, above all, isomorphic transition from the JG approved by the Customer to a program. Here, programming is performed in a unique way. Unapproved modifications of the algorithm are unacceptable.

In applying the technique proposed, it is possible to provide an agreement between the program text and the order of its execution, and to implement the procedure of stepwise refinement in accordance with the requirements of structured programming [31] together with the use of concepts of *object* and *class*, as in the conventional object-oriented programming [32].

2.6. Algorithmic Programming Languages

As is clear from the presented above, high-level and, especially, low-level algorithmic languages are appropriate in solving problems of logical control only at the stage of isomorphic transition from the automaton description to a program text, because, otherwise, many of the above mentioned problems arise, as applied to the use of problem-oriented languages for primary description.

3. JUMP GRAPHS AS A SPECIFICATION LANGUAGE

3.1. Strategies of Synthesis of Logical-Control Algorithms

There exist two strategies of construction of algorithms of this class. In applying the first of the strategies, it is supposed that the algorithm of operation of the control object is known, and it is necessary to synthesize (in terms of it) a logical–control algorithm that provides a given behavior of the object. As an illustration, we present a fragment of an algorithm for control of a valve synthesized in the manner shown above: "To open the valve, a computer must send a unit signal to the input of the opening actuator of the valve."

In the second strategy, taking the information of the state (position) of the controlling object, an algorithm that provides the required operation of the control object is constructed. For instance, "if the computer sends a unit signal to the input of the opening actuator of the valve, then the valve is opened."

The first strategy is "directed" from a control object to a computer and the second strategy has the opposite direction, i.e., from the computer to the object.

The first strategy is based on the concept of a state, and the second one is based on the concept of an event.

Presently, in the development of control algorithms, for example, in the form of graph diagrams of algorithms or in the form of productions (sequents) of the type "if...then" the second strategy is usually employed. Hence, in the author's opinion, the first strategy is more natural for the considered class of problems: a state by its nature is static and an event is dynamic, and, therefore, "control in states" is more advisable than "control in events" [33].

However, in the general case, neither of these two kinds of control is exhausting, and only "control in states and events" is correct here. Because the concepts of a state and an event incorporate the concept of an automaton, such kinds of control may be called automaton control, and its software implementation can be called automaton programming.

For the technique proposed in this paper, the concept of a state is primary and the notion of an event is secondary (especially, taking into account that events formed by a control object are dictated by the corresponding states of this object).

Although, the second strategy usually leads to the construction of faster and more compact programs, in the absence of rigid limitations on the memory capacity and performance the use of the first strategy is more natural, since it corresponds to the main principle of control employed in automated systems. This principle implies that, in control, the Operator defines a state of an object first and, then, executes a particular action that gives rise to an event.

Under this organization of the control process, for ease of reading and understanding of algorithms and programs, they must also be organized in the same way. The situation, when the control is organized in terms of one principle and its software is implemented on the basis of directly opposite ones is highly abnormal.

In the framework of the technique proposed, we must use automaton control and programming, and the construction of algorithms and programs has to start from generating a decoder of states, rather than from events [29], because in the control in events, essential

problems associated with the introduction and correct employment of internal variables arise.

In algorithmization, a state must be found based on an integral representation assigning to each state a decimal number considered as an indivisible component of description rather than on individual binary components.

If we assign to each state of the object the state of a controlling automaton, to which, in turn, a node of the JG is assigned, and construct the program that implements a jump graph of the automaton formally and isomorphically, then this program will be clear not only to the Developer and Programmer, but to the Customer, Technologist, Operator, and Inspector as well.

It should be noted that in spite of the complexity of the construction of a model of the object, in most cases, this model can also be described by jump graphs. Therefore, in the framework of the approach proposed, to verify the constructed algorithm, it is also expedient to simulate a complex CA–CO. After that, the control algorithm can be further refined by means of a physical model and using an actual object. The accuracy and detailedness of description of the algorithm by jump graphs increases dramatically the quality of initial algorithmization compared to other methods. That is why on an object, it is usually required that one introduce comparatively few modifications into the algorithm developed and the corresponding program.

3.2. Factors that Restrict Wide Applicability of Jump Graphs as Algorithmization Language

To eliminate disadvantages of the considered algorithmization languages, it is proposed to use as this language the jump graphs proposed more than forty years ago for describing the behavior of automata with memory. Jump graphs are also called state diagrams or state-jump diagrams.

However, in synchronous hardware implementation of automata, this language was mainly applied for illustrations, since in most optimization algorithms of the theory of automata a table representation of JGs, namely, transition tables and transition and output tables were employed. The structure of these tables requires the enumeration of all combinations of values for all input variables. This implies certain limitations on the dimension of the problems solved.

Another problem, which restricts the use of JGs, was associated with the fact that in the asynchronous circuit implementation of logical control systems because of the contradictions between the memory elements and an arbitrary way of change of sets of the input variables, an actual behavior of the circuit may significantly differ from the behavior of the model (jump graph), by which the circuit was built. In the general case, this calls for very cumbersome guiding coding associated with an excessiveness, which is often unacceptable for hardware and, especially, for switching implementation.

There is also another reason for this. Conventionally, it was supposed that JGs describe only sequential algorithms, which are of highly limited application for control systems characterized by parallelism.

The specified difficulties, as well as the traditions of construction of functional diagrams in a hardware and graph-diagrams in a software implementations of algorithms, are, evidently, the reasons why JGs have not been essentially used up so far as a specification language of operation conditions in their software implementation.

3.3. Jump Graphs

The basic concept employed in the theory of automata is an internal state of an automaton. Below, instead of this term we use the term "state." This concept, which is one of primary importance in human behavior (healthy–ill, full–hungry, and so on), for certain reasons is not usually employed in the algorithmization and programming of control processes (except for the approach employed in object-oriented analysis [34] and program products S7-HiGraph technique software [21] and Modicon State Language [22]).

Under other approaches usually, the internal state of Computer is either ignored or is not considered as the entire thing. Note that, as in event–controlled programming, only external events are checked and the actions initiated by these events are performed as, for example, this takes place for the following description of the algorithm: "If the table is set, then Computer must have its dinner." This description is implemented by a memoryless automaton, because, in this case, both the event and the action are observable only from outside.

This example demonstrates that, as a rule, for correct description of an algorithm, external information (the table is set) is insufficient, and that it is also necessary to know the internal state of Computer (whether it is full or hungry). The description takes the following form: "If Computer is hungry and the table is set, then Computer must have dinner." At first glance, it seems that the condition has been just quantitatively complicated, and instead of one variable we have two variables. However, this is not the case: the situation is completely different, since, now, the algorithm has the internal variable that must be in the Computer memory. Actually, it is unknown when this variable will be requested (most likely, when the table is set). Hence, together with the "combination diagram," the internal memory appears, and the "diagram" goes into the class of automata with memory.

The use of JGs makes it possible to introduce the concept of a state (in the explicit, namely, the graphic form) into practice of algorithmization and programming at least for problems of logical control along with its use in the theory of finite-state automata, linear sys-

tem theory [35], Markov processes [36], and in certain problems of practical [37] and theoretical [38] programming. JGs also allows one to represent in a descriptive form the dynamics of jumps of automaton from one state to another under a change of the input factors with indication of values of all the output variables generated at each state—for automata without output transformer (AOT), or Moore automata (MOA), or at each jump (for Mealy automata (MEA)).

If, in an automaton, both the methods of generation of output variables are used, then it is called "mixed" (M-automaton (MA)). If the same variables are employed in an automaton as the input and output variables, then this automaton is called an automaton with flags [29].

It should be noted that the same sets of values of the output variables may be formed in different states, and this requires the introduction of additional (intermediate) variables for distinguishing these states.

The automaton states classify (decompose) all input variables into groups selecting, in each of the states only the subset (of these variables) that specifies the required jumps from the state considered to be adjacent states including the jump into itself. The input variables that do not belong to the group specified by the state have no effect on jumps from it into the other states, i.e., the jumps from the state considered unessentially depend (do not depend) on all other variables that do not belong to the group. This provides an opportunity to implement problems of a large dimension by JGs.

Located at a certain state, an automaton with memory turns into a corresponding memoryless automaton (combination automaton). The latter automaton, according to the values of the output variables "chosen" by this state, chooses one of the adjacent states to which the considered one belongs. The new state "adjusts" the automaton to the implementation (in the general case) of another combination automaton. Thus, an automaton with memory may be considered as a multifunctional module that is adjusted by states to the implementation, in a certain order, of different orthogonal systems of Boolean formulas that describe combination diagrams and depend on different groups of input variables.

On the other hand, an automaton with memory may be considered as a multifunctional unit adjusted to implementation by input variables (whose values are usually constant during a program cycle) of self-governing (without input variables) automata.

If there exist values of input variables chosen for which the automaton retains its state, then this state is called stable. Otherwise, the state is called unstable.

Nodes correspond to states of automaton JG, and connecting arcs correspond to jumps from one state to another (the nomenclature of components of a JG is minimal). Note that a loop corresponds to the unchanged state, in which an automaton remains as long as the condition that labels the loop is fulfilled. The absence of a loop demonstrates that a node is unstable. An automaton (A) may be at an unstable node only for one program cycle. Usually, arcs are labelled by Boolean formulas of the input and time (X and T) variables. Unit values of these formulas define the possibility of jumps. Let us call attention to the fact that even a node having a loop can be unstable if the values of the formulas that label one of its entering points and one of its outgoing arcs that is not a loop are simultaneously equal to one.

The values of the input and time (Z and t) variables are specified in an explicit (bit) form at nodes (for AOT and MOA), on arcs (for MEA), and at nodes and on edges simultaneously (for MA). The binary variables t control functional delay elements (FDEs), and the binary variables T inform us about the operation or nonoperation of these elements. We assume that FDEs are not incorporated into automata and provide for one of the control objects. The complex A–FDE forms a unified component called a controlling automaton.

In the reading of a JG, it is supposed that at each instant (in one program cycle) no more than one jump is performed: zero if a state is retained and one if it is not retained. This is provided by a corresponding software implementation.

In each strongly connected JG employed for logical control, one source node that coincides with the terminal node is chosen.

3.4. Construction of Transition Graphs

Assume that it is required to construct a JG that describes the behavior of an automaton with n binary inputs and m binary outputs. For problems of a large dimension, a JG is usually constructed in accordance with a verbal description of operating conditions of the control object. In this description, the concept of state, which is a mathematical abstraction, of course, is not employed.

At the same time, if the formalization is performed by JGs of Mealy automata most commonly used in the literature for the description of examples of automata of a small dimension, then it is necessary to introduce this concept (abstraction) for distinguishing situations associated with *changes* of values of the output variables for the same values of the input variables. The construction of a JG in "changes" poses the further difficulties of their reading (understanding) and correction, because, here, the values of the output variables depend not only on the state, but on the values of the input variables essential for the considered state as well.

If state is defined as a combination of values of the *all* m output variables, and if the same combinations of these variables are considered as different states, then this definition is significantly less abstract and more natural, since the concept of state, as it were, in the explicit form is not used. In this case, it is possible to have, at *every* state, information on the value of *every* output variable. In the author's opinion, this is the gov-

erning factor what provides for further considerations of simplicity of reading, understanding, and correction of JGs.

At initial stages of construction of a JG, the number of points in it must be chosen, so that each of them supports one of the states of the control object (including states associated with its faults) or the control system (for example, the states that correspond to improper actions of the Operator). Below, if necessary, we may try to minimize the number of nodes. Note that, in the general case, the number of states in an automaton may be less than the number of states of an object because the same state of the automaton may support, by the same values of the output variables, different states of the object. For instance, a closed and opened state of a valve "with memory" (its stable states) can be supported by both a two and one state of the automaton with zero values of the output variables.

Each combination of all output variables corresponds to one point of a JG and labels it. Each node of the JG is directly connected by arcs with those of its nodes into which the automaton must jump if conditions that label the arcs are satisfied. Note that even neighboring nodes may be labelled in the same way. In this case, a condition is matched by a Boolean formula and fulfillment of the condition is matched by an equality of this formula to one with certain input sets. Note that only one Boolean formula corresponds to each arc. This formula depends on a subset of the input variables that semantically define jumps from the considered node into the adjacent ones, rather than on the all n input variables. This allows us to construct JGs for problems of a very large dimension.

When using this technique of construction, given operation conditions are implemented by a jump graph of an automaton without an output transformer with the forced [29] coding of states. Below, if necessary, this graph, for example, given if there are nodes labelled identically and there are no distinguishing input sets, is transformed into a JG of an automaton of another type with the explicit introduction of the concept of state (the JG of a Moore automaton) or another type of coding (the JG of automaton without an output transformer with the forced–free [29] coding of states). In this transformation the structure of the initially constructed jump graph remains the same. In particular cases, the account for limitations requires us to use a JG of a mixed automaton. This leads one to introduce, into a JG of an automaton without an output transformer or into a constructed by it JG of Moore automaton the values of the output variables (by ratio with a Boolean formula labelling the corresponding arc) generated in the jump. It should be noted that a JG may always be represented as a composition of a system of Boolean formulas of a memoryless automaton (formed by the formulas labelling arcs of the JG) and the jump graph whose edges are labelled by single letters and which replaces the corresponding formula.

3.5. Description of Operation of Memoryless Automata

The application of the presented technique allows one to implement operating conditions, no matter whether they correspond to an automaton with memory or to a memoryless one. For automata with memory, the construction of a JG in algorithmization is advisable, since it reflects in an explicit form at least the state-dependence (inherent for automata of this class) of the set of the output variables. For memoryless automata dependence of this sort is absent, and the values of the set of the output variables at the instant considered depend only on the set of the input variables at the same instant. Therefore, for these automata there is no need for application of JGs. Hence, if it is possible to find out whether a JG describes a memoryless automaton or can be reduced to it, it is appropriate to replace the JG, for example, by a system of Boolean formulas.

3.6. Properties of Transition Graphs

One of the advantages of JGs is that they can be formally tested for syntactical correctness (of course, the semantic (sense) correctness cannot be tested formally). A JG is correct if it is consistent, complete, and free of generating circuits different from loops.

In addition, we assume that the consistency of a JG is provided in the case if simultaneous jumps along any two or more arcs outgoing from the same node are forbidden in it. If simultaneous jumps from the same node are admissible, then such JGs (at each of them, the simultaneous existence of several "active" nodes is admitted) are called jump graphs with parallelism (PJGs). PJGs differ from GRAPHSET-diagrams only by the absence of a possibility to synchronize within a single component of completion of parallel processes. In the use of a system of interconnected JGs and a description of each algorithm by a particular JG (by a component), the synchronization of processes, if necessary, is performed in the head jump graph. The behavior of a PJG (dictated by the graph of accessible labelling, which is the graph of jumps between all the possible states of a component or a system of components) differs from its description—the PJG produces more than its structure describes.

From this standpoint, JGs (without flags and defaults that change depending on the prehistory of the values of the output variables at one node or on one arc of the graph) are analogous to parallel–serial contact circuits, for each of which the Boolean formula that describes its structure, simultaneously, defines its operation (behavior). Note that PJGs are analogous to bridge contact circuits, for which a Boolean formula that describes the behavior of each circuit does not specify its structure.

Although, for JGs of this type, concepts of node and state are synonyms, for PJGs, these concepts are not equivalent.

This feature of PJGs, which allows us to describe and implement in a compact and clear form certain classes of parallel processes by a single component (since, otherwise, one has to construct and implement a JG with a greater number of states), is connected with a loss of the most important property of a JG without flags and defaults. This property consists in a possibility to implement a JG, so that in the graph that describes the behavior of "implementation," the number of nodes is the same as in the source JG.

In formal (and correct) transition from the JG to the program text, this property may be fully retained for some methods of realization and partially retained for the others. The partial retainment of this property means that, e.g., in description of a given JG (in the case where the number of its nodes is not equal to a power of two) by SBFs and the reverse construction of a JG basing on this system, in a new graph nodes may appear that do not belong to the source representation. However, since there are not jumps from the nodes of the given JG to the new ones, despite the presence of jumps from new points to the given ones, this implementation is correct.

In implementation of JGs, for instance, by the *switch* construction of the C programming language, the specified property in view of their isomorphism can be completely retained [29].

In an informal transition from a JG to a program, the behavior of the program may differ from that of JG, perhaps, in such a way that using the JG as a test for the program, it is impossible to reveal their incompatibility. This is because in the JG the label of practically every jump is independent of some variables from the whole set of input variables that in the program with errors can be essential for the considered jump. For example, if a jump in a JG is executed for the label x4, and in the program the label x4&1x5 corresponds to this jump, then this error may be detected only accidentally without the exhaustive search or construction of the JG in terms of the program. This supports once again Dijkstra's statement that tests can detect new and newer errors in the program, but it is impossible to prove that after testing that the program has no errors. That is why this paper pays considerable attention to the construction of algorithms and programs that are "clear" to Specialists of various specialities. This must allow us to eliminate many errors in these products as a result of coordination at different design stages including the initial ones. Formality and isomorphism of construction of a program based on "clear" specification, for which the graph of attainable labelling is also constructed (and corrected, if necessary), leads to the fact that a JG may serve as a tool of program certification, rather than a debugging facility.

The consistency of jump graphs (conjunction of labels of any two arcs outgoing from the same node is zero) is guaranteed:

in the diverse arrival of contradictory values of variables;

in the operation with fronts of variables;

by orthogonalization (complication of labels) of contradictory edges (for example, in the implementation according to SBFs);

by the arrangement of priorities (the order of arrangement of program instructions in implementation by a method different from construction of SBFs is taken into account);

by the "splitting" nodes with contradictory arcs (the increase of the number of automaton states).

Completeness of the jump graph (disjunction of labels for all arcs outgoing from the node is equal to one) is verified after providing the consistency. In implementation of a JG by SBFs, all edges outgoing from each node have to be labelled, and for the other variants of execution of labelling loops for automata without an output transformer or a Moore automata may be defaulted. It is supposed that labelling a loop at a node guarantees its "completeness."

JGs have generating circuits if at least for one of them a conjunction of labels of all arcs that form is not zero. The elimination of generating circuits is executed by the same methods as the elimination of inconsistency (except for the arrangement of priorities).

3.7. Coding of States of Automata

To implement a JG its nodes must have different labels (codes).

If in a JG of AΦOT all nodes have different labels (the values of the input variables), then the same labels (as a whole, or particular components that distinguish the labels) can be employed as codes of states of the automaton. This method of coding is called forced.

If in a JG of AΦOT labels of certain nodes coincide, then to distinguish them, we introduce the least necessary number of additional (intermediate or internal) variables yi whose values distinguish identical nodes. This method of coding is called forced–free.

For Moore, Mealy, and mixed automata, we use free coding, for which codes of nodes of a JG are chosen independently of the values of the output variables associated with these nodes.

The "freedom" of coding for these types of automata in software implementation means that the codes for the chosen methods of coding in contrast to the asynchronous hardware implementation may be arbitrarily assigned to nodes of a JG.

Among all forms of free coding in software implementation of automata, it is most advisory to employ two forms of coding, namely, the binary and many-valued (integral) ones.

In the first case, we assign to the *i*th node of a JG a binary variable Yi that takes the unit value only in the *i*th node and zero value in the others.

In the second case, we assign to the entire *l*th JG a single many-valued variable YI whose *j*th value in turn is assigned to the *j*th node of the JG. This provides the implementation of algorithms with the *least possible number of additional variables.* It is this form of coding that provides the best reading of programs. As another advantage of this form, we highlight that it is not necessary to *clear* the previous value of a many-valued variable, since it is automatically performed in passing to another value of this variable. It should be noted that for a given single internal variable in one JG there is no competition of memory "elements," since this variable has no competitors.

Moreover, in correct organization of computations, there is no competition of memory elements for any form of coding. A program may implement a given algorithm either correctly or incorrectly, because, in contrast to a single asynchronous diagram it cannot behave in different ways depending on actual "elements' delay." The assignment of order of execution of program instructions is a kind of synchronization. For instance, if, as a programming language, we take the language of functional diagrams, and, on the basis of this language, we construct a circuit, then, in admissibility of changes of values of the input variables only at the beginning of a program cycle for one order of operation (numeration), this diagram has one completely determined behavior, and for another numeration it has another also completely determined behavior. In this case, neither behavior can follow the desired one.

Therefore, in program implementation, for any form of coding states that employs also not adjacent sets of variables, in formal transition from a JG to a program, this program in a "slow mode" (after a single computation under the program) operates in accordance with the JG despite the fact that in a "fast mode" (during a single computation) the values of variables may differ from the desired ones. In this case, in contrast to asynchronous circuits, difficulties do not arise, because intermediate values of each variable calculated within a program cycle are filtered.

For example, assume that in a direct jump from the state of a JG with code 00 to the state with code 11 the switching process is 00–10–11, where intermediate value 10 is filtered. The transition through the state with code 01 in program implementation based on SBFs is impossible (in contrast to asynchronous circuits), since the order of changing the values of variables is uniquely determined by the order of arrangement of formulas in the system.

3.8. Characteristic Properties of Application of Jump Graphs

In implementation of an algorithm by a single JG of the Moore automaton or the automaton without an output transformer and *formal* transition to a program text according to a jump graph without flags and defaults (this graph simultaneously is a graph of accessible labelling that completely describes the behavior of the automaton), this graph can also be a *test* for verifying the program. If the program is constructed based on the JG not only formally, but *isomorphically* as well (the descriptive equivalence between the JG and the program text is guaranteed), then testing can be replaced with the comparison of the program text with the JG.

If a JG has flags and defaults, then, to analyze its behavior in detail, we have to construct the graph of attainable labelling. This graph can be used as a certification test of the program.

Interaction of JGs in a system of interconnected JGs (SIJGs) can be performed in terms of the input, output, and, especially, many-valued internal variables that code nodes of graphs. This provides good obviousness and eliminates the necessity of application of additional internal variables for this purpose. A control algorithm can be represented as a head and called graphs, as well as in the form of components operating in parallel. SIJGs can also be constructed based on the embedding principle.

At nodes (as in GRAPHSET-diagrams) and on arcs of JGs, it is possible not only to assign and clear the output binary variables but to also start processes described, for example, by both JGs and AGDs as well. In the description of a process happening at a point of a JG with the help of AGD, it is repeatedly executed while the automaton stays at this node and terminates (after the termination of a recurrent pass of the AGD) after a jump of the JG into a new node. The execution at nodes of the graph of FDE jumps by means of a pulse variable that takes unit values every second gives an example of such a process.

For analysis of behavior (of all functionalities) of an arbitrary system of jump graphs, even if each graph has no flags and defaults, we have to construct a single (if all JGs of the system are interconnected) or several (if the system has separate groups of JGs) graphs of attainable labelling.

3.9. Basic Stages of Algorithmization in Application of Jump Graphs

The diagram of connections information sources (ISs)–controlling automaton–tools for information representation (AIR)–actuators (ACs) is developed.

A controlling automaton is decomposed into an automaton and FDE. This allows us to eliminate time from a model; only bit variables *t* required to start FDE (the outputs of the automaton) and bit variables *T* indicating an operation of these elements (the inputs of the automaton) are employed. The previous diagram is converted into the diagram of connections ISs–A–FDE–AIR–ACs.

If necessary, the automaton is heuristically decomposed into a system of interconnected automata (SIA)

of a lower dimension. The decomposition is performed by modes, objects, or in a mixed way.

The construction of the diagram of connections ISs–SIA–FDE–AIR–ACs completes the stage of the architectural (system) design.

Considering the ith automaton, along with functional delay elements controlled by it as the ith controlling automaton, we can assume that the last diagram contains a system of interconnected controlling automata.

For each automaton, we choose a structural model (a combination automaton, an automaton without an output transformer, a Moore automaton, Mealy automaton, and so on). Then, we code states of automata with memory.

Constructing a correct jump graph that uniquely corresponds to the chosen structural model and variant of coding states, for an automaton with memory incorporated into each controlling automaton and combining the graphs constructed into a system, we obtain a SIJG, which is a formal specification or a control algorithm. At this stage, formal specification for FDEs and models of control objects are also constructed. Construction of specifications completes the second stage of design of a control program.

At the third stage of design (if necessary), we choose, construct, and optimize algorithmic models that implement formal specification taking into account the type (the first or second) of the structural model of each automaton with memory (for instance, the Moore automaton of the second type [39]).

The proposed technique includes methods of formalized passing from JGs to different types of algorithmic models. The main of them are systems of Boolean formulas, functional diagrams, ladder diagrams, and AGDs without internal feedbacks.

Clearly, JGs are also algorithmic models, for which it is also necessary to know the type of the structural model chosen before programming.

The choice of one or another algorithmic model depends on the programming language. Certain languages, e.g., C are applicable for any model considered. For other languages, like functional diagrams, the number of such models is restricted to one.

After the choice of algorithmic models for formal specifications, we go to the last (the fourth) stage of the technique proposed, namely, programming. At this stage, after the choice of a programming language, for each algorithmic model, we choose a software model that specifies, among other things, the list of admissible operators.

A detailed example that illustrates the proposed approach is presented in [40].

3.10. Programming

In using each programming language, a program formally constructed based on a JG (directly or on the basis of other models) may be isomorphic or not isomorphic (by its structure) to the JG, based on which way it was constructed.

In the first case, the *proof* of the equivalence of the program to the corresponding JG can be obtained by their comparison. In this case, the JG can always be reconstructed directly based on the program text without additional computations.

In the second case, the program is difficult to read, but there is no need for this, because the program is *formally* constructed on the basis of the JG, which should be of interest alone. In this case, the program is tested on the basis of the JG by the "scheme": present state, the input–next state, and the output. The verification based on construction of the JG is essentially more cumbersome in this case and connected with computations. It is natural that the isomorphism between the program text and the JG is provided at the cost of excessiveness (it is impossible under rigid limitations on memory capacity).

The main peculiarity of the proposed software implementation is that in a program cycle, the program executes no more than one jump in the JG, since, in the opposite case, if, for a certain node of the JG of the Moore automaton, conditions that label one of the entering and one of the outgoing (except for a loop) arcs are satisfied, then the values of the output and internal variables that must be formed at this node are filtered (dropped).

The proposed technique is called after the *switch* construction of C programming language since it allows one to most simply pass from the constructed JG to the text of the program isomorphic to it in structure. In this case, the operator *switch*, which provides a multivariant (many-valued) choice, guarantees the decomposition of automaton with respect to its internal states.

3.11. Software and Methodical Support of the Technique

A possibility of fast, error-free, and isomorphic going from a JG to a program text in a high-level language, e.g., C, makes it possible to simplify considerably the debugging and modeling of a controlling automaton and, if necessary, of a complex CA–CE, describing not only the automaton, but of the model of the control object (by components and/or modes) based on JG as well.

Under the author's supervision, a program shell, that allows one to change by a keyboard (on a personal computer for SIJG involving N jump graphs implemented in C) the values of the input variables and to watch on a display the values of the output, temporary, and, above all, N internal variables (only by *one* internal

variable for each JG) in step-by-step and automatic modes, is developed.

The possibility of a reading on the display of the decimal number of state for each JG in each program cycle by only a single many-valued variable makes the program completely *observable and controllable.* This fundamentally distinguishes the proposed technique from the conventional techniques of programming. Using the conventional techniques, it is always possible to display (by a debugger) any variable and to keep watch on changes of their values. However, in this case, it is difficult to answer the following questions:

What variables should be used in the program to ensure its effectiveness and controllability?

How many variables (especially, internal) should be used?

What is characterized by each variable?

What set of variables should be displayed at each stage of debugging?

What variables should be additionally introduced into the program and displayed if at a certain stage of debugging its workability has not yet been guaranteed?

Solving these problems for tasks of considered class is considerably simplified, if we already at the stage of developing the algorithm and/or its fragments introduce (in a regular way) into them states instead of introduction of individual variables (in an irregular way) that represent components of states in the entire process of program development.

If we employ a high-level algorithmic language such as C, then after constructing an algorithm by the technique proposed including its certification and modeling by the shell mentioned, the development is completed. Using the other programming languages, at the next stages, we formally (manually or automatically) pass (perform a synthesis) from the program developed to the program text in the language employed.

For example, for the PLC Autolog of FF-Automation, B.P. Kuznetsov (NPO Avrora), in cooperation with the author of this paper, developed a C translator–ALPro that makes it possible to obtain automatically (using the text of the structured program written basing in the JG in a subset of C) a program in the ALPro language of instructions with translation controlling constructions chosen in advance, namely, a conditional jump or step register [3].

With constraints on internal resources of PLC, the author developed methods of "manual" formal implementation of a JG on the basis of ALPro. Methods of implementation of JG on the basis of other programming languages, like ladder and functional diagrams [39], are also developed. One of these methods makes it possible, among other things, to construct functional diagrams isomorphic to JGs on the basis of elements for the Selma-2 system [41].

4. FINAL RECOMMENDATIONS

The proposed approach makes it possible:

(1) To apply the theory of finite-state deterministic automata in algorithmization and programming of control processes.

(2) To describe initially desired behavior of a controlling "device" rather than its structure, which is secondary and, therefore, more difficult for reading and understanding.

(3) To introduce into algorithmization and programming the concept of state (as a basic one) to begin algorithmization from determination of the number of states.

(4) To introduce the concept of automaton programming and automaton program-development.

(5) To begin the construction of automata and programs from forming a decoder of states rather than events.

(6) To use the basic structural models of the theory of automata and to introduce new ones.

(7) To use as an algorithmization language jump graphs and systems of interconnected jump graphs.

(8) In construction of jump graphs to eliminate the dependence on "deep" prehistory of states and the outputs and, if possible, the dependence on the values of the input variables of automata with memory from the output variables of automata with memory.

(9) To employ many-valued coding of states for each jump graph and to use independently of the number of its nodes only one internal variable that codes them.

(10) To use, as the basic algorithmic model, jump graphs of the Moore automata whose codes of states and the output values are radically separated, and the values of the output variables in each state are independent of the input variables, which simplifies modifications.

(11) To provide realization of a number of properties of control algorithms such as composition, decomposition, hierarchy, parallelism, invoking, and nesting.

(12) To have a single specification language in different programming languages, including specialized ones employed in programmable logical controllers.

(13) To perform algorithmization as a result of contacts between the Customer, Technologist, and Developer. The preparation of the specifications for the project become a single process of contacts that are completed by construction of the JG or a system of interconnected JGs taking into account details with an accuracy up to each state, jump, and bit, rather than a single event with "endless" consequent supplements.

(14) To use JGs without flags and defaults, whose number of nodes coincides with the number of states of the automaton, as a certification test and to construct a checking graph or a graph of attainable labelling for the other classes of jump graphs or systems of intercon-

nected jump graphs to check their behavior (this allows one to replace testing of programs by the analysis of their functionalities).

(15) To introduce a formal criterion of clearness of various forms of descriptions of automata with memory, namely, the isomorphism of each automaton with the corresponding JG without flags and defaults.

The automaton given by this JG may be called an absolutely white box and the automaton given in any other form is called a relatively white box.

An automaton for which the maximum number of states in its minimum form is known (this automaton can be recognized on the basis of analysis of the "input–output" sequences) is called in [42] a relatively black box, an automaton, whose internal content is unknown (this automaton cannot be recognized in the specified way), is called in [4] an absolutely black box. Hence, if the concept of state in algorithmization and programming is not introduced, then in response to the testing of algorithms and programs by input–output sequences, in the general case, it is impossible to recognize automata implemented by these algorithms and programs.

(16) To employ methods of formal and isomorphic passing from the specifications to programs of logical control in various programming languages.

(17) Applying high-level algorithmic languages to program by using the *switch* constructions (including nested ones) or analogous to them. In addition to the isomorphism with the specification, this provides accessibility of each value of the many-valued variable that codes states of each jump graph, for the other graphs of the system. Therefore, this eliminates the necessity of additional internal variables for the interaction of graphs.

(18) To introduce into programming the concept of observability (this makes it possible to consider a program as an absolutely white box, in which all internal variables, whose number is minimal, are accessible for observation).

(19) To test programs in two stages: first, to check the availability of all jumps provided by JG analyzing numbers of states and, second, to check statically at each state the values of the input variables.

(20) To assist Participants of a design (the Customer, Technologist (Designer), Developer, Programmer, Operator (User), and Inspector) to understand uniquely and completely what should be done, what is done, and what has been done in the functional part of the software implemented, i.e., to solve for the class of problems considered a problem of mutual understanding.

(21) To take into account the technical specifications in detail at early stages of development and to demonstrate to the Customer the way it is understood.

(22) To divide the work and, above all, the responsibility among the Customer, Technologist, Developer, and Programmer. This is particularly important if specified Specialists are representatives of different organizations and, especially, countries, since, in the opposite case, essential linguistic and, in the end, economic problems arise.

(23) To provide contact for Participants not routinely aware of terms of technological processes (for example, a mode of special start does not "go") in an intermediate completely formalized language (technical Esperanto of a sort). One may speak for instance, in the following way: "In the third JG at fifth node at the fourth position, change the value from 0 to 1." This does not cause confusion that could arise through an ambiguity of understanding even in one natural language (and, especially, for several languages in the case where Participants are representatives of different countries) and does not require the participation of Specialists that can make sense of technological processes for a correct introduction of changes [43].

(24) To make it possible for the Programmer not to know peculiarities of a technological process and for the Developer not to make sense of details of programming.

(25) To make it possible for the Application Programmer not to think up something for the Customer, Technologist, and Developer, but only to implement formally and uniquely a system of JGs in the form of a program. This allows us to reduce considerably the requirement on his professional skill and, in the end, to give up his services at all and employ computer-aided programming or computer-aided programming by the Developer. However, the last-mentioned way may be used only in the case where programming is "open" for the Developer, whereas this does not necessarily take place especially if one works with foreign firms or their entities that produce control systems, and not only hardware.

(26) To leave clear "traces" upon completion of development. This allows new persons to modify programs (in the traditional approach such modification is very labor-consuming, because "it is easier to develop a new program than to unravel somebody else's"); in this case, it should be noted that structuring and comments solve this problem only partially.

(27) To simplify modifications of the specifications and program and to improve their "reliability."

(28) To make control algorithms invariant to programming languages. This gives us an opportunity to form and support libraries of algorithms written strictly and formally.

(29) To provide an opportunity for the Customer, Technologist, and Developer to verify texts of functional programs, rather than to check the output, as it is common practice nowadays.

(30) To eliminate differences in unequal "reliability" of formally testing the hardware and software by the Inspector, since, in the first case, besides the operation, he checks many other characteristics (for instance,

the quality of printed circuit boards as well as their coating, quality of soldered joints, values and designation of elements, and so on) and, in the second case, attention is focused only on the check of operation, whereas the internal organization of programs and technique of their development are not studied.

5. PRACTICAL APPLICATION

The approach proposed was applied, specifically, to the following:

In constructing a system of control of DGR-2A 500★500 Diesel-generator for a vessel of project no. 15640 on the basis of Selma-2. The programing is carried out in NPO Avrora in the language of functional blocks [2, 41].

In constructing a system of control of a Diesel-generator of the same type for a vessel of project no. 15760 in cooperation with Norcontrol (Norway). The program implementation was carried out by the firm in PL/M [43].

In constructing the complex system of control of hardware components for a vessel of project no. 17310 on the basis of PLC Autolog. The program implementation was carried out by NPO Avrora in ALPro manually (for general-ship systems) and automatically by using a translator (for systems of control of auxiliary devices of the propulsion engine).

CONCLUSIONS

While the proposed and introduced in 1991 (by the author) switch-technique was developed [41, 44], a very strange situation arose. The leading firms in the automation field (except for [22]) completely ignored techniques of algorithmization and/or programming based on jump graphs, which are the foundation of the considered methodology. Only in 1996, this situation fundamentally changed: Siemens, in addition to its software, developed a new product called S7-HiGraph technique software, which allows one to use state diagrams (another name of a transition graph) in programming languages. This development gives an opportunity to automatically generate executable (only PLCs of this firm) codes, according to the description in the language.

In the opinion of the officials of Siemens the description in this language is not only appropriate for the PLC Programmer, but is clear to the Mechanical Engineer, Equipment Engineer, and Service Engineer as well [21].

Especially strange in the cited situation is that such a product could be developed, for example, fifteen years ago, since state diagrams were proposed more than forty years ago. However, world passion for Petri nets, which provided the means for describing parallel–serial processes by one component (functional unit) and, therefore, represented the basis of GRAPHSET (S7-Graph technique software), most likely, psychologically did not allow this to happen.

The author hopes that after the advent of the product in question, a "domino effect" will take place and in the immediate future many firms of the world will produce analogous products.

It should be noted that [39] technique, which proposes the use of jump graphs and systems of interconnected graphs not only as a programming language, but as a specification language for problems of logical control in employment of any other programming languages (for example, C or C++) as well, is described in detail, can become a useful manual for employment of controlling graphs in algorithmization and programming for a wide class of Users of industrial computers and programmable logical controllers (for example, [45]), including the cases where translators from this specification language are lacking.

The discussed technique is an essential supplement to International Standard IEC 1131-3 [46], in which programming languages of PLCs are described, but methods of algorithmization and programming (described in detail in [39]) are not presented.

The proposed technique may become the basis for an increase in safety of software for systems of logical control [47].

This technique does not rule out other methods of building "error-free" software [48], but it is significantly more constructive, since it allows one to begin "debugging" even at the stage of algorithmization.

The use of the pentad: state–independence of deep prehistory–system of interconnected jump graphs–many-valued coding–construction switch (or its analog in any programming language, for instance, in a language of functional diagrams) provides obviousness, structuredness, invoking, nesting, hierarchy, controllability, and observability of programs as well as their isomorphism (descriptive equivalence) with specifications, by which they are formally constructed. This makes it possible for the Customer, Technologist (Designer), Developer, Programmer, Operator (User), and Inspector to understand each other uniquely and know exactly what must be done, what is done, and what has been done in software implemented design. This also makes it possible to divide the work, as well as the responsibility, and, in addition, easily and correctly introduce modifications into algorithms and programs.

Thus, it can be stated that, by analogy with a theory of switching diagrams and probability theory, in this paper it is proposed to construct parallel–serial and Markov programs, which are easy to read and understand, instead of bridge and non-Markov programs.

In conclusion, we note that, in [49] it is proposed to use jump graphs in software implementation of algorithms of logical control of technique processes. However, an approach presented there was insufficiently "elegant" and this prohibited its wide practical use.

The employment of systems of interconnected jump graphs is considered in the works of V.V. Rudnev (for instance, [50]). However, their theoretical nature and the use of binary variables for the connection of graphs limit the implementation of this model in actual practice.

The author hopes that the fate of the proposed approach in terms of its practical implementation will be more successful.

REFERENCES

1. Dijkstra, E.W., *A Discipline of Programming*, Englewoodcliffs [USA]: Prentice-Hall, 1976. Translated under the title *Distsiplina programmirovaniya*, Moscow: Mir, 1979.
2. "Selma-2." *Description of Functional Blocks*, ABB Stromberg Draivs, 1989.
3. *Autolog 32. User Guide*, FF-Automation, 1978.
4. *Programmable Controller. MELSEC-A. Programming Manual. Type ACPU. Common Instructions*, Tokyo: Mitsubishi Electric, 1979.
5. Pospelov, D.A., *Situatsionnoe upravlenie. Teoriya i praktika* (Case Control: Theory and Practice), Moscow: Nauka, 1986.
6. Butakov, E.A., *Metody sinteza releinykh ustroistv iz porogovykh elementov* (Methods for Synthesis of Relay Devices Based on Cut-Off Elements), Moscow: Energiya, 1970.
7. Karpovskii, M.G. and Moskalev, E.S., *Spektral'nye metody analiza i sinteza diskretnykh ustroistv* (Spectral Methods for Analysis and Synthesis of Discrete Devices), Moscow: Energiya, 1973.
8. Malyugin, V.D., Implementation of Boolean Functions by Arithmetic Polynomials, *Avtom. Telemekh.*, 1982, no. 4.
9. Artyukhov, V.L., Kondrat'ev, V.N., and Shalyto, A.A., Implementation of Boolean Functions by Arithmetic Polynomials, *Avtom. Telemekh.*, 1988, no. 4.
10. Field, A. and Harrison, P., Functional Programming, Translated under the title *Funktsional'noe programmirovanie*, Moscow: Mir, 1993.
11. Baranov, S.I., *Sintez mikroprogrammnykh avtomatov (graf-skhemy i avtomaty)* (Synthesis of Microprogram Automata (Graph-Schemes and Automata), Leningrad: Energiya, 1979.
12. *Mistic Controller. Opto. Booklet N1-800-321-OPTO*, Tokyo.
13. Nassi, J. and Shnenderman, B., Flow Chart Techniques for Structured Programming, *SIGPLAN Not*, 1973, no. 8.
14. Lyapunov, A.A., On Logical Schemes of Programs, in *Problemy kibernetiki*, vol. 1, Moscow: Fizmatgiz, 1958.
15. Bardzin', Ya.M., Kalnin'sh, A.A., Strods, Yu.F., *et al.*, *Yazyk spetsifikatsii SDL/PLUS i metodika ego ispol'zovaniya* (SDL/PLUS Specification Language and the Methodology of Its Application), Riga: Latvia Gos. Univ., 1986.
16. Vel'bitskii, I.V., *Tekhnologiya programmirovaniya* (Technique of Programming), Kiev: Tekhnika, 1984.
17. Peterson, J., *Petri Net Theory and the Modeling of Systems*, Englewoodcliffs [USA]: Prentice-Hall. Translated under the title *Teoriya setei Petri i modelirovanie sistem*, Moscow: Mir, 1984.
18. Yuditskii, S.A. and Machergut, V.Z., *Logicheskoe upravlenie diskretnymi protsessami* (Logical Control of Discrete Processes), Moscow: Mashinostroenie, 1987.
19. Mishel', Zh., *Programmiruemye kontrollery. Arkhitektura i primenenie* (Programmable Controllers: Architecture and Application), Moscow: Mashinostroenie, 1992.
20. Berger, G., *Programming of Controlling Devices in the STEP-5 Language. Vol. 1. Programming of Basic Functions*, Siemens, 1982.
21. *SIMATIC. Simatic S7/M7/C7. Programmable Controllers*, SIEMENS. Catalog ST 70, 1996.
22. *Modicon Modsoft. Programmer User Guide*, GM-MSFT-001, 1993.
23. Gavrilov, M.A., Devyatkov, V.V., and Pupyrev, E.I., *Logicheskoe proektirovanie diskretnykh avtomatov* (Logical Design of Discrete Automata), Moscow: Nauka, 1977.
24. Ambartsumyan, A.A., Iskra, S.A., Krivandina, N.Yu., *et al.*, A Task-Oriented Language of Description of Systems of Logical Control, in *Proektirovanie ustroistv logicheskogo upravleniya* (Design of Logical Control Devices), Moscow: Nauka, 1984.
25. Devyatkov, V.V. and Chichkovskii, A.B., The Uslovie-82 Language of Program–Logical Control, in *Avtomatizatsiya proektirovaniya. Vyp. 2* (Automatization of Design. Vol. 2), Moscow: Mashinostroenie, 1990.
26. Gorbatov, V.A., Smirnov, M.I., and Khlytchev, I.S., *Logicheskoe upravlenie raspredelennymi sistemami* (Logical Control of Distributed Systems), Moscow: Energoizdat, 1991.
27. Kuznetsov, O.P., Shipilina, L.B., Markovskii, A.V., *et al.*, Problems of Development of Logical Programming Languages and Their Implementation on Microcomputers (On the Example of the Yarus-2 Language), *Avtom. Telemekh.*, 1985, no. 6.
28. Kuznetsov, O.P., Graphs of Logical Automata and Their Transformations, *Avtom. Telemekh.*, 1975, no. 9.
29. Shalyto, A.A., Use of Graph–Schemes of Algorithms and Graphs of Switchings in Program Implementation of Algorithms of Logical Control, *Avtom. Telemekh.*, 1996, no. 6, 7.
30. Varshavskii, V.I., Rozenblyum, L.Ya., Marakhovskii, V.B., *et al.*, *Aperiodicheskie avtomaty* (Aperiodic Automata), Moscow: Nauka, 1976.
31. Linger, R., Mills, H., and Witt, B., *Structured Programming*, Reading [USA]: Addison–Wesley, 1979. Translated under the title *Teoriya i praktika strukturnogo programmirovaniya*, Moscow: Mir, 1982.
32. Buch, G., *Ob"ektno-orientirovannoe proektirovanie s primerami primeneniya* (Object-Oriented Programming with Examples of Application) Kiev: Dialektika, Moscow: AO "IVK", 1992.
33. Matcho, D. and Folkner, D., *DELPHI*, Moscow: BINOM, 1995.
34. Shleer, S. and Mellor, S., *Ob"ektno-orientirovannyi analiz. Modelirovanie mira v sostoyaniyakh* (Object-Oriented Analysis: World Simulation in States), Kiev: Dialektika, 1994.

35. Zade, L. and Dezoer, Ch., *Linear System Theory*, New York: McGraw-Hill, 1963.

36. Gnedenko, B.V., *Kurs teorii veroyatnostei* (Probability Theory), Moscow: Nauka, 1965.

37. Kas'yanov, V.N. and Pottosin, N.V., *Metody postroeniya translyatorov* (Methods of Design of Compilers), Novosibirsk: Nauka, 1986.

38. Martynyuk, V.V., On Analysis of Graph of Switchings for the Operator Scheme, *Zh. Vych. Matem. Matem. Fiz.*, 1965, vol. 5, no. 2.

39. Shalyto, A.A., *SWITCH-tekhnologiya. Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (SWITCH-Technique: Automatization of Programming Problems of Logical Control), St. Petersburg: Nauka, 1998.

40. Shalyto, A.A. and Antipov, V.V., *Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (Algorithmization and Programming of Logical Control Problems), St. Petersburg: Morintekh, 1996.

41. *Project 15640. AS 21. DG1. CONTROL. AMIE. 95564.12M*, St. Petersburg: ASS Avrora, 1991.

42. Trakhtenbrot, B.A. and Bardzin', Ya.M., *Konechnye avtomaty. Povedenie i sintez* (Finite Automata: Behavior and Synthesis), Moscow: Nauka, 1970.

43. *Functional Description. Warm-Up & Prelubrication Logic. Generator Control Unit. Severnaya Hull N431. Norcontrol*, 1993.

44. Shalyto, A.A., Program Implementation of Controlling Automata, *Sudostroitel'naya Promyshlennost'*, *Avtom. Telemekh.*, 1991, no. 13.

45. *Modicon-Telemecanique. TSX Micro Industrial Programmable Controllers. Groupe Schneider, Catalog*, 1996.

46. International Standard IEC 1131-3. Programmable Controllers. P. 3. Programming Languages, *International Electrotechnical Commission*, 1993.

47. Underwriters Laboratories Renews the Safety Standard for Software for PLK, *Sovremennye tekhnologii avtomatizatsii*, 1997, no. 1.

48. Baber, R.L., *Programmnoe obespechenie bez oshibok* (Software without Errors), Moscow: Radio i Svyaz', 1996.

49. Gorbatov, V.A., Kafarov, V.V., and Pavlov, P.G., *Logicheskoe upravlenie tekhnologicheskimi protsessami* (Logical Control of Industrial Processes), Moscow: Energiya, 1978.

50. Rudnev, V.V., A System of Mutually Connected Graphs and Simulation of Discrete Processes, *Avtom. Telemekh.*, 1984, no. 9.

1. Translated under the title, Teoriya lineinykh sistem, Moscow: Nauka, 1970.

Рец? М.б. сократить науч. ж-лов? (3р)

Пер? Дж. Уайлд (2р)