Automation and Remote Control, Vol. 57, No. 6, 1996

AUTOMATA

ALGORITHMIC GRAPH SCHEMES AND TRANSITION GRAPHS: THEIR USE IN SOFTWARE REALIZATION OF LOGICAL CONTROL ALGORITHMS. I

A. A. Shalyto

UDC 519.687

The difficulties encountered in understanding graph schemes used in designing algorithms and programs are examined. A classification of graph schemes is given. The main difficulties in understanding graph schemes have been shown to result from the negligence of a concept like "state." If this concept is introduced, transition graphs can be used instead of graph schemes. Requirements for transition graphs are formulated, which, when satisfied, make them "meaningful."

1. INTRODUCTION

Presently, mainframes are more often used in combination with a programmable logical controller in software realization of logical control algorithms for technological processes.

Irrespective of the type of control computer used, since the problems (e.g., controlling nuclear reactors) of this class are vital, it is imperative that all interested parties, i.e., customer, designer, programmer, operator, and controller, understand each other completely without confusion.

We call the language of communication between them the language of specifications [1]. Although a large number of such languages are available for logical control programs [2–26], due to fully formed traditions in the field of computer technology [5], block schemes, which are also known as graph schemes or simply schemes [27, 28], find extensive use in practice. These schemes are strongly recommended by national standards [29, 30].

These standards, however, only define the rules for imaging graph schemes and do not specify any requirements (except for imaging) on the construction of these schemes that might aid in understanding them.

Literature, too, does not fully describe the properties of algorithmic graph schemes that might aid in using them as a communicative language for this class of problems. For example, methods of constructing graph schemes are designed in [31, 32]; their structural organization improves the understanding of these schemes. The authors believe that if graph schemes are designed only from basic controlling constructions without the use of GOTO operators, the graph schemes yield to easy understanding. But they did not pay attention to other factors that render graphs hard to read, especially the omission of values of variables.

Recently, program designers have realized that structural design alone is not adequate for solving this problem [33]. Therefore, an object-oriented approach incorporating the concepts of an "object" and its "state" to program designing was developed. This approach stipulates the use of transition graphs for describing the dynamics of processes being realized. Except for one isolated example illustrating the use of transition graphs, [33] does not describe any theoretical principles governing the requirements to be imposed on the construction of transition graphs so that programs could yield to easy understanding.

In this paper, we elaborate the requirements for constructing easily understandable graph schemes and transition graphs. In Part II of this paper, we shall design methods of constructing understandable transition graphs from algorithmic graph schemes and understandable algorithmic graphic schemes from transition graphs, as well methods of programming in languages of different levels.

Avrora Scientific Production, Inc., St. Petersburg, Russia. Translated from Avtomatika i Telemekhanika, No. 6, pp. 148–158, June, 1966. Original article submitted March 10, 1995.

 0005-1179/96/5706-0890$15.00

2. GRAPH SCHEMES. BASIC PROBLEMS

In this paper, we study automatic graph schemes in which only unit or zero values of variables are formed and stored at operator vertices. Automatic graph schemes are subdivided into two classes: algorithmic graph schemes and software graph schemes.

Furthermore, we differentiate algorithmic graph schemes by the following criteria:

(a) internal feedback,
(b) types of variables,
(c) state decoder,
(d) omission of variables and ambiguously defined variables,
(e) variables which repeatedly vary in one pass through a graph scheme, and
(f) output destination for the values of output variables.

On the basis of these criteria, without claiming completion of the classification, we distinguish five subclasses of algorithmic graph schemes:

AGS1, i.e., algorithmic graph schemes with internal feedback, in which the values of output variables are sent to any operator vertex,

AGS2, i.e., algorithmic graph schemes with no internal feedback and with state decoder, in which values of output variables are delivered at the end of the "body" of a graph scheme,

AGS3, i.e., algorithmic graph schemes with no internal feedback, in which the specific properties of the controlling constructions of the programming language are taken into account,

AGS4, i.e., algorithmic graph schemes with no internal feedback but with a state decoder, in which the values of output variables are delivered at the end of the "body" of a graph scheme and which do not contain any output variables varying repeatedly in one pass over graph schemes, and

AGS5, i.e., algorithmic graph schemes with no internal feedback but with a state decoder, in which the values of output variables are delivered at the end of the "body" of a graph scheme and which contain output variables that may vary repeatedly in one pass over graph schemes.

It must be noted that external feedback is always inherent in control algorithms and control programs (scanning mode in programmable logical controllers). It must also be noted that if the control algorithm is realized in the computing device as a component (of the problem), then both AGS1 and AGS2 can be isomorphically mapped into a software graph scheme through the use of AGS3 with appropriate controlling constructions.

But, if the control algorithm is realized in the computing device as several components, then there is no possibility of isomorphically mapping an AGS1 into a software graph scheme. This happens because in logical control problems solved through prolonged use of AGS1 under certain conditions the procedure gets into an internal feedback loop, and this hinders the realization of other control algorithm components. Looping may take place, for example, until a " key is pressed," "an alarm fails," "a time limit is exhausted," or "a shaft executes a few revolutions."

By way of example, Fig. 1 illustrates the block diagram of the connection between a controller and an object. It consists of three channels Ch1, Ch2, Ch3, and a motor M. The controller is composed of an automaton (A) and delay units (DU). Figure 2 shows an AGS1 with five internal feedbacks for this example [34]. This graph scheme realizes either the controller as a whole (DU1 and DU2 are computed in the third and fourth operator vertices, or only the automaton A (time variables t_1 and t_2 are regarded as inversions of the procedure implementing time delay units, whereas the binary variables T_1 and T_2 warn the actuation of delay units).

This graphic scheme is not complete, because it does not show the operator vertices into which the output variables z_i $(i = 1 \ldots 4)$ and time variables t_j $(j = 1, 2)$ are dumped. Therefore, it cannot be used as a formal specification of the problem and needs to be further elaborated. Paying attention to the principles of operation of single-input servos used in the controlled object, we can assert that these servos are not provided with memory and therefore in AGS1 (Fig. 2) the value of a variable at those operator vertices where no variable is shown is, by default, zero.

This assumption that the value of an output or time variable at an operator vertex is zero if the variable is not contained in the vertex is not always true, because in using a graph scheme it is often assumed that omitted variables retain their previous values. Such graph schemes can be used in computations, since the computer remembers the previous values of all variables stored in the external memory. But man cannot remember the prehistory, especially the values of several variables concurrently. It must, however, be noted that graph schemes are primarily designed to map control connections and, secondarily, to depict data [28, 35]. Therefore, graph schemes with neglected values of variables are not useful as a communicative language.

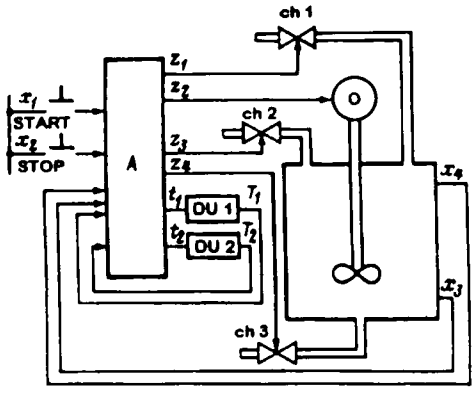

Fig. 1

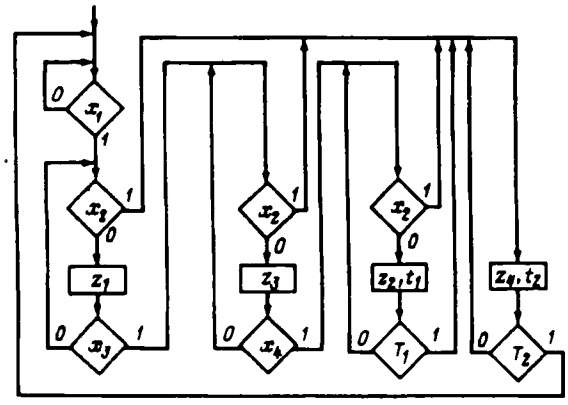

Fig. 2

Only a specification that describes the control connections and depicts data equally to the maximum extent can be regarded as a quality specification [35]. Because of the absence of certain data in explicit form, a graph scheme cannot be used as a test for verifying the validity of a program implementing this graph scheme. Moreover, if the program is heuristically designed from graph schemes, even in the absence of omissions, we can only verify whether a program realizes a given graph scheme, but it is not possible to establish that the program does not execute anything else additionally.

As regards the specific properties of different subclasses of algorithmic graph schemes, we must note that in the schemes of the subclass AGS1 the values of output variables are dumped not at the end of the body of the scheme, but at any operator vertex.

The schemes of the subclass AGS1 can be designed such that only input variables of the types X and T are delivered at conditional vertices, while output variables of the type Z and t are dumped at operator vertices, where T denotes variables representing the actuation of delay units. The main merit of the schemes of the AGS1 subclass is that they contains only those variables that are used in the control algorithm.

In the AGS2 subclass, the variables that are stated in the control algorithm are rarely used, and "superfluous" variables are not used at all. Figure 3 shows, by way of example, an AGS2 implementing an R-trigger, in which only the variables defined in the control algorithm are used, namely, trigger setting variable x_1, trigger reset variable x_2, and output variable z. Even this AGS (without any additional variables) is difficult to understand, because the results are delivered at the end of the program body, and therefore for $x_1 = x_2 = 1$ the values of z are computed twice (recomputed). Furthermore, for $x_1 = x_2 = 0$, this scheme contains a route having no operator vertices. In passing over this route, the preceding value of z is saved; this is achieved by storing the values of z at the operator vertices in an external memory cell.

This example shows that if an AGS2 contains even one route in which neither the zero value nor the unit value of at least one output variable is not set, such a graph scheme implements a sequential automaton; otherwise it realizes a one-cycle automaton.

As already mentioned, schemes of the subclass AGS2 with no "superfluous" variables are rarely encountered. In general, in the schemes of this subclass, in addition to the values of input variables of the types X and T, the values of the output variables Z at conditional vertices are also verified. Intermediary (internal) variables Y not present in the control algorithm, which are set and dumped along with output variables at operator vertices, are also checked. Since binary variables Y are often used, AGS2 schemes are rather unwieldy. Furthermore, they, as a rule, are structurally not ordered, because the layout of vertices and their labels in AGS are not governed by standards. It is rather difficult to understand the schemes of the subclass AGS2 for the simple reason that the stored values of the variables Y and Z are neglected.

It is also difficult to understand the schemes of the subclass AGS2 if the values of variables depend on their prehistory (values of the corresponding variables stated at the preceding operator vertices). These schemes are especially difficult to read if the values of variables not only depend on the prehistory, but also vary with the prehistory, i.e., depend on the routes leading to the operator vertices. In the latter case, serious difficulties arise in recording the changes occurring in graph schemes without errors.

While it is quite natural for a programmer to verify the variables Y and Z and meaningful for a designer, the presence of these variables for a customer, an operator, or a controller is unacceptable, because the customer usually

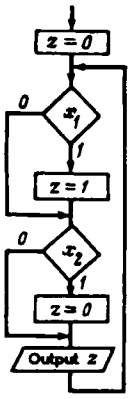

Fig. 3

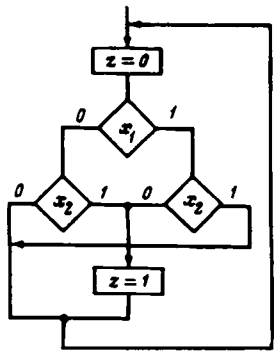

Fig. 4

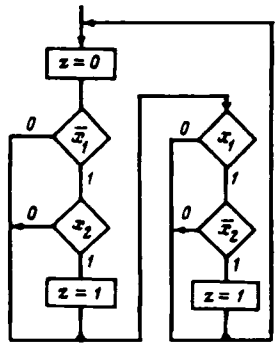

Fig. 5

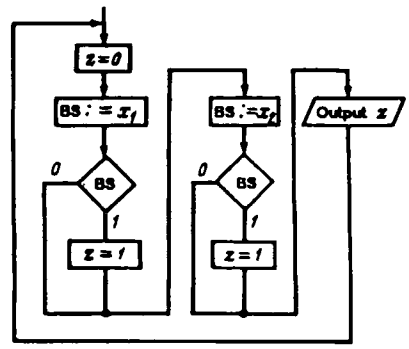

Fig. 6

does not require the unit value for any internal variable in specifications. Serious reading difficulties may arise if the variables Y and Z in an AGS2 scheme vary under the action of other control algorithm components realized in the same computer.

From the foregoing, it follows that while the schemes AGS1 and AGS2 in which the variables Y and Z are not verified at conditional vertices show only the semantics of the control algorithm, the schemes AGS2 with such verification procedures are indeed algorithms for computer-aided realization of control algorithms. Therefore, such graphic schemes not helpful as a communicative language.

The construction of schemes of the subclasses AGS1 and AGS2 does not end here, because an AGS3 must be constructed from the initial algorithmic graphic scheme with regard for the basic properties of the controlling constructions of the programming language to achieve errorless transition to software graphic schemes and for consultation (for vital objects). For example, in most programmable logical controllers, for the IF command, only forward transitions are allowed, whereas backward transitions are prohibited. Another specific feature of the IF command in many programmable logical controllers is that they are, first, single-address controllers, and, second, they possess a unique property, namely, if an IF command transfers the control to the CONT command (flag) when appropriate conditions are satisfied, then all other IF commands occurring between these two commands must also transfer the control to the flagged CONT command when the conditions are satisfied [34]. Moreover, the unconditional transition STOP command aids in realizing only external feedback.

Under these conditions, the schemes of the subclass AGS3 must be linearized and structured only with controlling constructions "sequential union" and "incomplete selection." If the initial AGS2 possesses such a structure (see Fig. 3), there is no need to construct an AGS3. But, if an AGS2 implementing a single-cycle automaton described by one of the Boolean formulas $z = \bar{x}_1 \& x_2 \vee x_1 \& \bar{x}_2 = x_1 \oplus x_2$ has a "planar" (Fig. 4), but not a "linear" structure, then we must preliminarily construct an AGS3 in order to transform the algorithmic graph scheme to a software graphic scheme and the program code (Fig. 5). The last graphic scheme is more difficult to read than the scheme AGS2 (Fig. 4) (even though the variables Y and Z are not verified in these graphic schemes), because input variables are to be repeatedly verified in passing over any route in AGS3. Nevertheless, if the Boolean formulas are programmed in operator form (directly from formulas), the scheme AGS2 will always have a linear structure. This, indeed, in the general case will reduce operation speed and require the use of intermediary variables.

In computing devices in which input variables are fed as needed (as in certain programmable logical controllers), in every pass over an algorithmic graph scheme, these variables are not only verified repeatedly, but they vary their values as well. This may violate the functional transformation defined in the specification (truth table). By analogy with hardware realization, this violation can be called the program realization risk. Risk in such programmable logical controllers can be reduced through bifurcation or maximal nonrepeated realization.

Thus, while an AGS1 depicts only the semantics of the control algorithm, and AGS shows, in addition to semantics, the possibility of realization of control algorithms consisting of several components in one computing device, AGS3 reveals the specific features of the controlling constructions of programming language as well. But such an AGS3 is still far being a software graphic scheme, because the latter must also show the semantics of all applied commands or operators of the programming language. Furthermore, references to those architectural features of the computing device that are not used in the algorithmic graph scheme appear in software graphic schemes. For example, Fig. 6 shows a software graph scheme for implementing an AGS2 (Fig. 3), in which BS stands for binary summator of the programmable logical device. From the foregoing, it follows that software graphic schemes, especially program codes, must not be used as a communicative language.

Therefore, only AGS1 without omissions can be a candidate as a communicative language. But problems related

to delooping, linearization, and structurization are encountered in programming AGS1. For this reason, AGS2 are constructed in favorable cases and, avoiding the construction of other graphic schemes, program texts are written in an informal form.

Additionally, we encounter the problem of selection of texts and proof of the validity of the program, because, since there is no "distinct" specification, it is not possible to judge whether a problem is understandable or not. Moreover, the testing the "validity" of a program is shifted off to experiments which reveal what is wrong in a program. But it is rather difficult to detect whether a program, if it contains bugs, can do anything that is not defined in the specification.

This problem is further complicated by the fact that in practice the usual method of verifying the operation of a program is represented as a two-column table showing the values of input and output. But such a verification, which is appropriate for single-cycle automata, is not suitable for sequential problems. Because of the unwieldly large dimension, it is rather doubtful whether a test could be designed that could take account of the values of all internal variables, as well as their preceding values in certain cases, especially if the algorithmic graphic schemes are structurally disordered.

In my opinion, all these problems mainly result from the fact that the concept of "internal state" of a component as a whole is not used in algorithmic graphic schemes, software graphic schemes, and programs; only separate binary variables that characterize the internal state in an indirect manner are used in them. In the sequel, we refer to this internal state as simply "state." Introduction of this concept will permit us to pass on from algorithmic graphic schemes to transition graphs and vice versa, and to use transition graphs as a communicative language and specification.

3. TRANSITION GRAPHS. CLASSIFICATION

Definition 1. *A transition graph (TG) in which the values of every input variable are specified at every vertex is called a Moore machine (MM) transition graph with explicitly defined values of all output variables.*

In this case, the output values correspond to the number of the vertex and do not depend on the prehistory. For this reason, such a transition graph is easy to understand and changes are easily entered in it.

Definition 2. *A transition graph at whose vertices, along with explicitly defined values of some variables, implicitly but uniquely defined values of other variables are also used is called a Moore machine (MM) transition graph with implicitly defined values of output variables.*

Implicit specification of a variable at a vertex is indicated by a dash, which represents only one value of a Boolean variable. The variable at a vertex retains the value assigned to the variable at all "adjacent" vertices. Connection between two vertices can be a direct arc interconnecting the vertices or a transit arc through other vertices in which dashes are used to denote the values of this variable.

As regards reading, transition graphs of this type are somewhat worse than the MM transition graphs of the first kind, but these transitions graphs are helpful in reducing memory space occupied by a programs which isomorphically maps MM transition graphs. Transition graphs of this class can be mapped such that every vertex explicitly shows the values of every output variable, and the values of these variables which vary at "adjacent" vertices are labelled by a cross.

Definition 3. *A transition graph in which the explicitly defined values of some variables and implicitly but uniquely defined values of other variables are used at vertices is called an MM transition graph with ambiguous specification of the values of output variables.*

Transition graphs of this class may contain a fewer number of vertices for a problem than the MM transition graphs of other two types mentioned above, because at a vertex, instead of one value of a neglected variable, different values of a variable may be formed and, consequently, different sets values of output variables may be formed at this vertex. Thus, one such vertex may become equivalent to certain definite vertices.

This seeming merit from the viewpoint of compact description and program size reduction leads to serious shortcomings of the transition graphs of this type—the graphs are difficult to read and understand. Precisely for this reason, such transition graphs are not useful as a communicative language.

Specifications for the problems of this class are best defined through the first two MM transition graph models, whereas, if AGS1 is used in specification, AGS1 must be transformed into one of these MM transition graph models.

This, however, does not exclude the use of transition graphs for other automaton classes. Analogous classification can be made for the Mealy machine (MeM) transition graphs, in which the values of output variables are

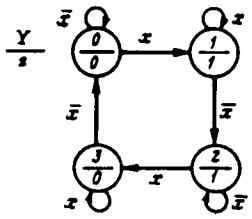

Fig. 7

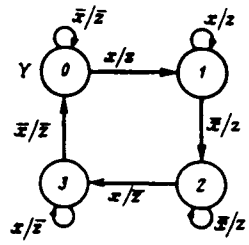

Fig. 8

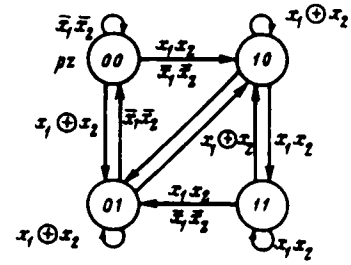

Fig. 9

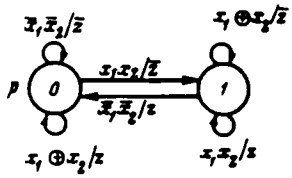

Fig. 10

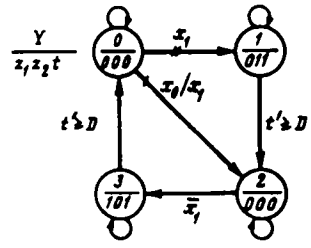

Fig. 11

formed not at graph vertices but on arcs [36]. In many logical control problems, the number of states (number of vertices in the transition graph) is not reduced in passing from a Moore machine model to a Mealy machine model. Figure 7 shows an MM transition graph implementing a flip-flop counter, while Fig. 8 illustrates this problem in terms of an MeM transition graph. In other cases, transition from MM transition graphs or from automaton transition graphs without output converters to MeM transition graphs reduces the numbers of vertices. Figure 9 shows the transition graph of a machine without output converter having four vertices that realizes a sequential one-digit summator. In Fig. 10, this algorithm is described in terms of a Mealy machine with two vertices.

It must be noted that while for Moore machines and machines without output converter with unique specification of the output values the number of states is equal to the number of combinations of these values (including repeated values), for automata in which all combinations are distinct the number of states is equal to the number of distinct combinations. For automata of these classes with ambiguous specification, the concept of "state" is loosely related to output values. Therefore, this concept for these automata is rather abstract and less comprehensible. This situation becomes rather complicated for the Mealy machines. In [37], for Mealy machines with ambiguous output values, the concept of "situation" was introduced instead of the concept of "state," and the term "transition graph" was replaced by the term "switching graph." Although a switching graph may contain a fewer number of vertices than an equivalent transition graph, a switching graph is not helpful as a communicative language, because this graph is rather difficult to understand.

An analogous situation may also be observed for the number of states due to the omission of values of the variables in mixed machines, i.e., "machines without output converter" (mixed Mealy machines of type 1), "Moore–Mealy machines," and machines of all these classes with flags, in which the same variable may be used in some transition graphs as an input variable as well as an output variable.

In several cases, variables can be used as a flag, provided they are stored in memory cells, their values are verified by an automaton and can be varied not only by some external data source, but also by the internal data source of the automaton. In Moore–Mealy machine transition graphs (Fig. 11), a variable x_1 can be used as a flag. The values of this variable are stored in an external binary memory cell and can be changed in stable states of the machine by some button or by the machine itself in the course of transition 0–2 induced by the input variable. The machine not only forms the values of the variable x_1, but also verifies them in the course of transitions 0–1 and 2–3.

Typically, flags are used as additional variables, which act at the memory cells with fixed contents, i.e., they cannot be changed by other data sources. For the flags introduced in machines without output converters, these memory cells may be regarded as integral components along with the cells containing the values of variables which differentiate states.

For Moore, Mealy, and mixed machines, as flags we can use additionally introduced variables F, whose values are stored in external memory cells. If multivalued coding is used to encrypt the states of a complex machine belonging to one of these classes, it suffices to have only one intermediary variable in the machine. In such cases, upon the

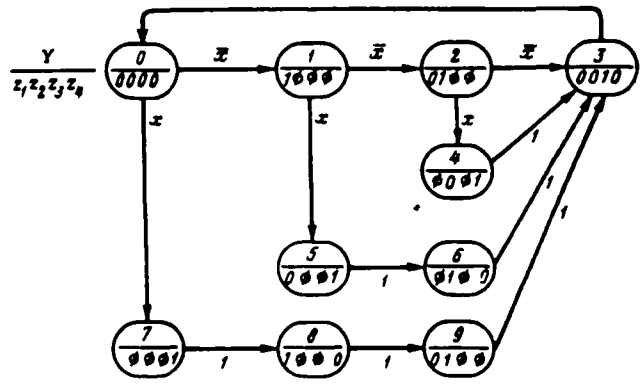

Fig. 12

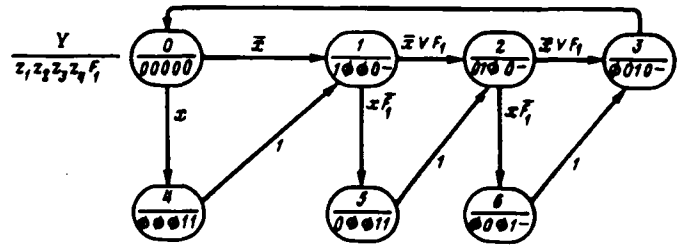

Fig. 13

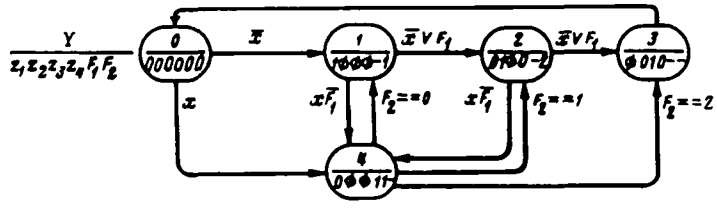

Fig. 14

introduction of flags in an external environment, additional memory cells, including multivalued cells, are generated. However, the number of succeeding states depends not only on the number of the state and the input action (as in the case of machines without flags), but also on the prehistory.

In these machines, not only the output but also the states may depend on and vary with the prehistory. Consequently, they cannot be easily read.

Figure 12 shows an MM transition graph with ten vertices, none of which is stable. The number of vertices can be reduced to seven by introducing a binary flag F_1, which is formed only by the machine, making the flag values at certain vertices ambiguous (Fig. 13). The number of vertices can be reduced further by introducing an additional multivalued flag F_2 and omitting its values (Fig. 14).

In Part II, we shall elaborate methods for constructing understandable transition graphs from algorithmic graph schemes and understandable algorithmic schemes from transition graphs, as well as methods of programming these graph models in languages of different levels.

REFERENCES

1. V. N. Agafonov, ed., *Requirements for Program Specifications* [in Russian], Mir, Moscow (1984).
2. S. Cleany, "Representation of events in neural networks and finite automata," In: *Automata* [Russian translation], Inostr. Lit., Moscow (1956), pp. 17–27.
3. A. A. Lyapunov, "Logical schemes of programs," *Probl. Kibern.*, **1**, 5–28 (1958).
4. Yu. I. Yanov, "Logical schemes of algorithms," *Probl. Kibern.*, **1**, 29–53 (1958).
5. L. A. Kaluzhnin, "Algorithmization of mathematical problems," *Problemy Kibern.*, **1**, 58–63 (1958).
6. A. A. Tal', "Enquete language and abstract synthesis of minimal sequential machines," *Avtomat. Telemekh.*, No. 6, 38–49 (1964).
7. M. A. Gavrilov, V. V. Devyatkov, and A. B. Chichkovskii, "Language of operator schemes of parallel algorithms with memory (OSPAM language)," in: *Abstract and Structure Theory of Relay Devices* [in Russian], Nauka, Moscow (1975), pp. 18–30.
8. V. M. Glushkov, Yu. V. Kapitonova, and A. A. Letichevskii, "Use of the method of formalized technical specifications in designing data structure processing programs," *Programmirovanie*, No. 6, 5–12 (1978).
9. O. P. Kuznetsov, "Theory of algorithmic finite-automata languages," *Avtomat. Telemekh.*, No. 3, 122–133 (1981); No. 4, 127–135 (1981).
10. O. P. Kuznetsov, L. B. Shipilina, A. V. Markovskii, et al., "Development of logical programming languages and their computer realization (exemplified by YaRUS-2 language), *Avtomat. Telemekh.*, No. 6, 128–138 (1985).
11. V. V. Devyatkov and A. B. Chichkovskii, "Condition-82—logical control language," in: *Computer-aided Designing* [in Russian], Mashinostroenie, Moscow (1990), Part 2, 58–67.
12. A. A. Ambartsumyan, S. A. Iskra, N. Yu. Krivandina, et al., "Problem-oriented language for describing the behavior of the FORUM-M logical control system," in: *Design of Logical Control Devices* [in Russian], Nauka, Moscow (1984), pp. 35–47.
13. S. A. Yuditskii and S. S. Pokalev, *Logical Control of Flexible Integrated Production* [in Russian], Preprint, Institute of Control Sciences, Moscow (1989).
14. A. D. Zakrevskii, *Logical Control Language* [in Russian], Preprint, Institute of Technical Cybernetics, Minsk (1988).
15. V. G. Lazarev and E. I. Peil, *Synthesis of Controlling Automata* [in Russian], Energoatomizdat, Moscow (1989).
16. S. I. Baranov, *Synthesis of Micro-Program Automata (Graph Schemes and Automata)* [in Russian], Energia, Leningrad (1979).
17. G. Florin, "Stage tables or Petri networks?," in: *Theory of Discrete Control Units* [Russian translation], Nauka, Moscow (1982), pp. 35–42.
18. V. N. Zakharov, "Sequential description of control automata," *Izv. Akad. Nauk SSSR, Tekh. Kibern.*, No. 2, 58–65 (1972).
19. G. Fraitag, V. Gode, K. Jacobi, et al., *Introduction to Techniques of Handling Solution Tables* [in Russian], Energiya, Moscow (1979).
20. D. Bazil, "Realization of finite automata in Fort language," in: *Fort in Research and Development* [in Russian], Leningrad University, Leningrad (1991), Vol. 1, Part 1, pp. 5–8.
21. G. Sosie, "Controlling automata: simulation, decomposition, and realization," in: *Theory of Discrete Controlling Units* [in Russian], Nauka, Moscow (1982), pp. 49–58.
22. Ya. M. Bardzin', A. A. Kalninsh, Yu. F. Strods, et al., *SDL Specification Language and Its Application Method* [in Russian], Latvian State University, Riga (1986).
23. G. Berger, *Programming of Controlling Devices in STEP 5*, SIEMENS (1982).
24. G. Mishel, *Programmable Controllers. Architecture and Application* [in Russian], Mashinostroenie, Moscow (1992).
25. A. A. Shalyto, "Software realization of controlling automata," *Sudostroitel. Promyshlennost', Ser. Avtomat. Telemekh.*, No. 13, 41–42 (1991).