Automation and Remote Control, Vol. 57, No. 7, 1996

AUTOMATA

ALGORITHMIC GRAPH SCHEMES AND TRANSITION GRAPHS: THEIR APPLICATION IN SOFTWARE REALIZATION OF LOGICAL CONTROL ALGORITHMS. PART II

A. A. Shalyto

UDC 519.714

Methods of constructing "readable" transition graphs from algorithmic graph schemes and constructing "readable" graph schemes from transition graphs are designed. Programming methods which guarantee readability of programs for languages of different levels are constructed. The proposed method is compared with the basic structural programming method—the Ashcroft–Mann method. The advantageous of the method are discussed, requirements for using a transition graph as a communicative language are formulated, and the specifications for the studied class of problems are stated.

1. INTRODUCTION

The properties of algorithmic and software graph schemes that make these schemes hard to understand are discussed in [1]. A classification of algorithmic graph schemes is proposed. When transition graphs satisfying certain conditions are used, specifications become readable and understandable.

In this paper, we design methods for constructing understandable transition graphs from algorithmic graph schemes, and also solve the converse problem, namely, construction of understandable algorithmic graph schemes from transition graphs. We shall present programming methods which guarantee understandability of programs written in languages of different levels that implement diverse functional problems of logical control. The proposed method is compared with the basic structural programming method—the Ashcroft–Mann method.

2. CONSTRUCTION OF READABLE TRANSITION GRAPHS FROM ALGORITHMIC GRAPH SCHEMES WITH FEEDBACK

We shall illustrate the method by an example. Assuming that an algorithmic graph scheme (AGS) (Fig. 1) with internal feedback (IFB) is given (in [1], this scheme is referred to as AGS1), we are required to understand this graph scheme (GS).

By digitizing (encircled numbers) the starting point and the end point (if any), as well as the points succeeding operator vertices (OV), and determining the route between adjacent points in the algorithmic graph scheme [2], we can construct a Mealy machine (MeM) transition graph with ambiguous values for output variables (Fig. 2), in which the number of vertices is equal to the number of points introduced in the algorithmic graph scheme. This transition graph can be effectively programmed, for example, in C, but it is not a simple matter to understand how it works (due to the omission of certain values of output variables) and, accordingly, how an AGS1 scheme functions.

Therefore, using AGS1 schemes, we shall construct a Moore machine (MM) transition graph, which must be easier to understand by its construction principles [1]. For this purpose, let us assign every operator vertex a number (without circles) (Fig. 1) and determine all possible routes between adjacent vertices [2]. Using this information, we shall construct an MM transition graph with ambiguous values of output variables and containing five vertices, each of which is assigned a multi-digit (decimal) number (Fig. 3). This transition graph is easier to understand than the previous graph, but is highly complicated due to the omitted values of variables.

AVRORA Scientific Production, Inc., St. Petersburg, Russia. Translated from Avtomatika i Telemekhanika, No. 7, pp. 144–169, July, 1996. Original article submitted March 10, 1995.

0005-1179/96/5707-1027$15.00 © 1996 Plenum Publishing Corporation

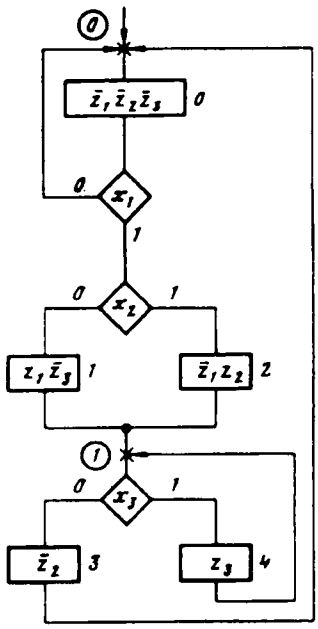

Fig. 1

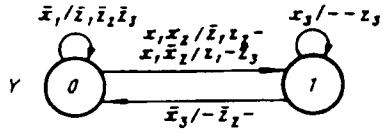

Fig. 2

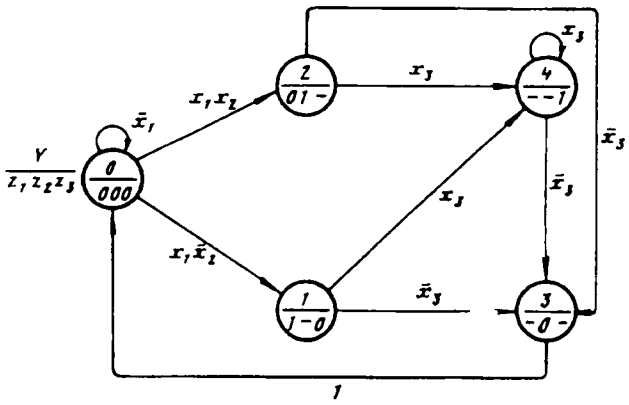

Fig. 3

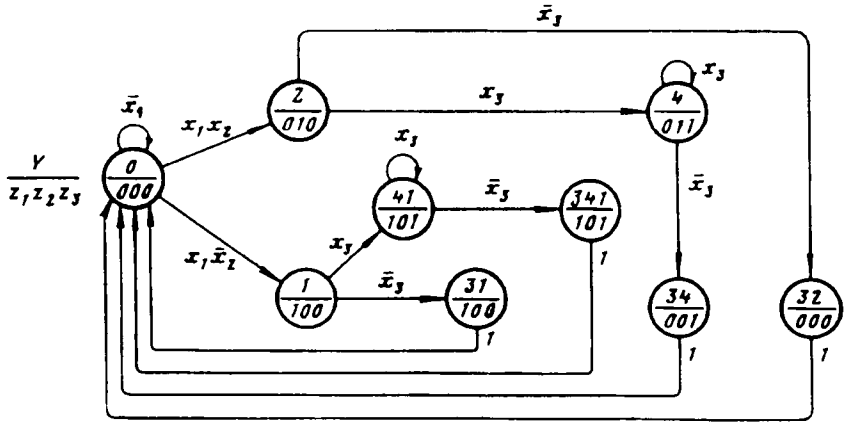

Fig. 4

Analyzing the values formed for different routes in this transition graph, we shall construct an MM transition with nine vertices but no omitted values (Fig. 4). Since identical values of output variables are formed at each of the two pairs of vertices (41, 341) and (1, 31), the second vertex in these pairs along with the arcs labeled with one (unconditional transition) can be excluded. The first vertices, however, in these pairs are joined to the zero vertex. The zero vertex is joined with vertex 2, because vertex 32 in which the same values of output variables are formed as at the zero vertex can be excluded. The MM transition graph thus obtained contains six vertices (Fig. 5). This transition graph is "absolutely" understandable, because it is compact and no knowledge of its prehistory is required to read it. However, its structure, contrary to the transition graph constructed in Figs. 2 and 3, is quite distinct from the initial algorithmic graph scheme.

This transition graph is full of contradictions, but contains, like the scheme AGS1, two generating circuits $0 = 1$ and $0 = 2$. Eliminating these circuits, for instance, by introducing a variable x_3 in the label of the arcs $0 - 1$ and $0 - 2$, we obtain a correct transition graph (Fig. 6). Such changes are easily introduced without any changes in the structure of a transition graph, whereas this needs considerable corrections in the structure of an algorithmic graph scheme and may lead to errors.

From the foregoing, it is clear that precisely transition graphs (Fig. 6), but not AGS1 schemes, must be used as a "means of communication" with customers and as a programming specification in the absence of any stringent constraints on the memory space to be occupied by the program. This transition graph, unlike any AGS1, describes

the behavior of automaton (A) in an explicit and understandable form, and contains adequate information so that the automaton can be formally realized by diverse algorithmic models (systems of Boolean formulas, functional schemes (FS), AGS4 [1], etc.) [3, 4].

Each of these algorithmic models, in turn, can be isomorphically represented by a program model, but the degree of isomorphism between program models and transition graphs is different. Indeed, while the codes of programs written, for example, in the C language and formally constructed by a system of Boolean formulas or a functional scheme are functionally equivalent to transition graphs, these codes on transition graphs are outwardly dissimilar, especially for a system of Boolean formulas describing a functional scheme with triggers. Nevertheless, this system of Boolean formulas is quite different from the system of Boolean formulas constructed directly from transition graphs. On the other hand, the language contains a control construction, like a switch, which ensures isomorphism between program codes and transition graphs, and this makes specifications and program codes easily readable.

Finally, let us note that while the information shown in Fig. 6 is essential for constructing certain algorithmic and software models, for example, a system of Boolean formulas or a functional scheme, the labels of loops can be omitted, provided a switch is used in the transition graph (under the assumption that the labels of arcs outgoing from a vertex are logically complete at each vertex), and we can define priorities, depicting them by dashes on arcs, instead of orthogonalizing the labels. The higher the priority, the lesser the number of dashes (Fig. 7). This, in the general case, will greatly simplify transition graphs and reduce the size of programs without worsening their understandability.

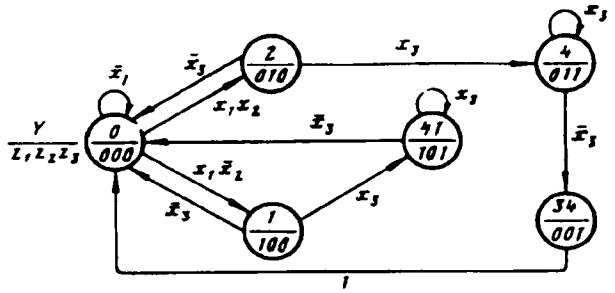

Fig. 5

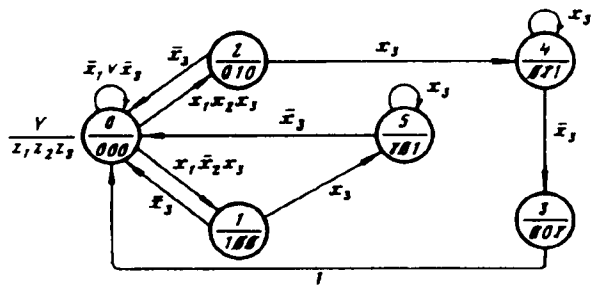

Fig. 6

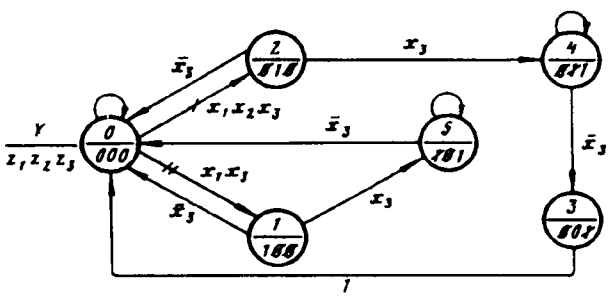

Fig. 7

Let us now examine the construction of a readable algorithmic graph scheme without internal feedback from a transition graph containing no omissions of output values. As demonstrated in [1], "badly organized" algorithmic graph schemes without feedback (in [1] referred to as AGS2) are hard to read. We now show what should be the structure of an algorithmic graph scheme without feedback so that it may be free of this drawback.

3. AUTOMATA WITHOUT OUTPUT CONVERTER BUT WITH FORCED STATE CODING

Let us study the transition graph of an automaton without output converter but with forced state coding (Fig. 8), in which there are no generating circuits for the pulse variables x_1 and x_2. The coding is called forced, because codes defining the states coincide with the values of output variables formed in the respective states (vertices). We use this type of coding for the reason that combinations of the values of output variables are distinct at the vertices in the transition graph. This transition graph is used in realizing a four-layered AGS4 [1] (Fig. 9). The first layer contains conditional vertices labeled by the variables z_1 and z_2, and is a state decoder. The second layer consists of conditional vertices labeled by input variables, and implements the transition functions of the automaton. The third layer contains operator vertices, which show the output values. These output values in this case coincide with the codes of succeeding states.

Although the values of output variables are not shown at the operator vertices, this graph scheme is easily understood, because, unlike AGS2, there is no need to save the values of stored values since they can be determined by "going upward" along an appropriate route in the graph scheme.

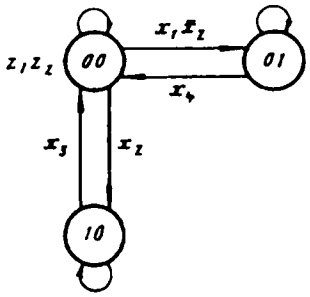

Fig. 8

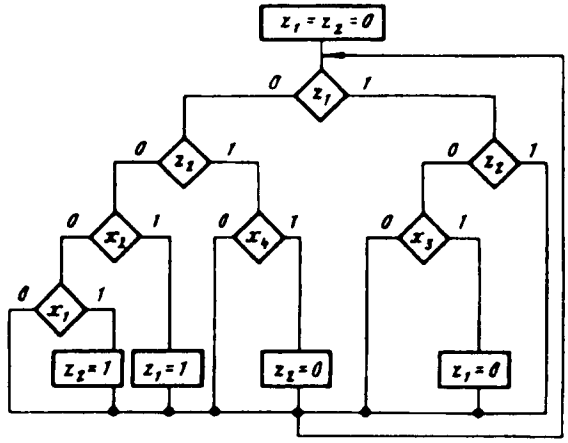

Fig. 9

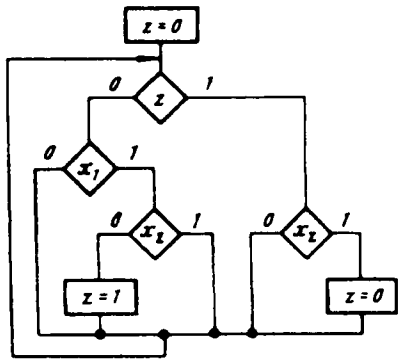

Fig. 10

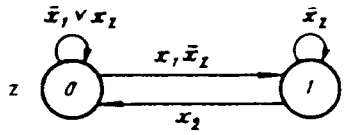

Fig. 11

This graph scheme is readable for one more reason: unlike AGS2 (Fig. 2 in [1]), it has depth, which is determined by only one transition in the transition graph. This is an extremely important and useful property of this class of graph schemes. Furthermore, in this graph scheme, unlike in the graph scheme of Fig. 3 in [1], there is no need to compute repeatedly the values of any output variable, and, unlike in the graph scheme of Fig. 5 in [1], there is no need to verify repeatedly the values of the same input variable. This graph scheme is not only structured in the sense of traditional meaning, but is also well organized in the sense that the conditional vertices with labels Z and X are not mixed up with each other or with operator vertices.

Such an organization of an algorithmic graph scheme (current state–transition condition–next state) is in agreement with human normal behavior—man, on getting up in the morning, first determines his inner state (alive or dead, healthy or ill), and only then "inquires" about the values of input variables (whether it is warm or cold outside), and finally, depending on the values of these variables, enters the next state (for example, getting dressed to match the weather). Obviously, behavior with regard for only the values of input variables, but with no regard for the inner state, would be rather odd.

But the concept of a "state" is usually not used in explicit form in algorithmizing through algorithmic graph schemes without internal feedback. The state is constructed, beginning from input variables and, then, depending on their values, we form the values of output variables and, possibly, additionally introduced internal variables, which determine the states indirectly (by the component method).

Although an AGS (Fig. 3 in [1]) implementing an R-trigger is in ideal agreement with the structural programming principles, it suffers from two drawbacks which make the scheme difficult to read: recomputation of the values of z for $x_1 = x_2 = 1$ and the absence of the values of z in explicit form for $x_1 = x_2 = 0$. These drawbacks are not inherent in an algorithmic graph scheme with state decoder (Fig. 10). This scheme, though it is highly complicated compared to the algorithmic graph schemes shown in Fig. 3 in [1], is easier to understand. The transition graph shown in Fig. 1 is simple to read and compact in representation. For programming, it can be simplified further (omission of loop labels) through the use of SWITCH constructs.

4. AUTOMATA WITHOUT OUTPUT CONVERTER BUT WITH FORCED-FREE STATE CODING

We now study the construction of a flip-flop counter, whose behavior is described by a transition graph (Fig. 12). In order to distinguish the vertices in the transition graph, we use forced-free state coding by introducing an internal variable y that is not present in the control algorithm (Fig. 13). This coding is called forced-free, because the values of some digits in the code are forcibly defined by the values of the output variable z, while the values of the other digits, which are denoted by the variable y, can be chosen arbitrarily in course of software realization. Figure 14 shows an algorithmic graph scheme, which is equivalent to this transition graph.

Since the states of the automaton A depend on the values of the output z in the algorithmic graphic scheme of Figs. 9, 10, and 14, difficulties may arise if the output values vary in the other components of the control algorithm. Therefore, using a Moore machine model, we shall exclude the dependence of the states of the automaton A on the output values.

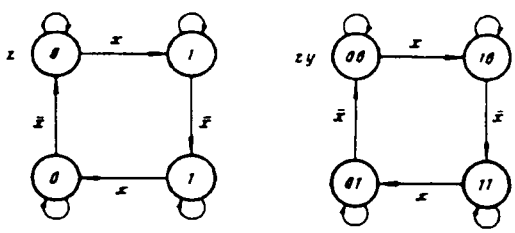

Fig. 12

Fig. 13

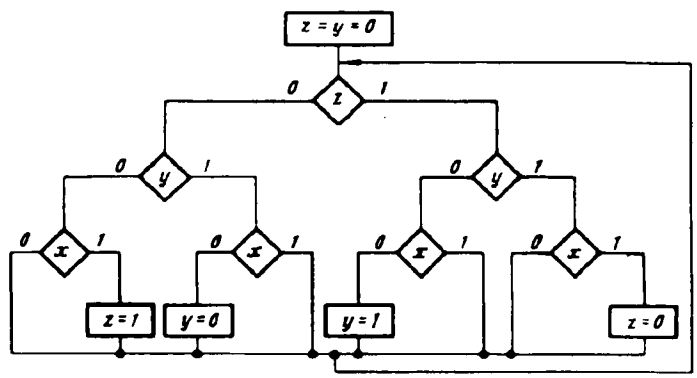

Fig. 14

5. MOORE MACHINES WITH BINARY LOGARITHMIC STATE CODING

Figure 15 shows a Moore machine transition graph, which implements a flip-flop counter for binary logarithmic state coding, while Fig. 16 shows the corresponding algorithmic graph scheme. This algorithmic graph scheme consists of four layers (state decoder, formation of output values, implementation of transition functions, and formation of the next state). Its drawbacks are that the pyramidal decoder is rather tall, and the number of newly introduced variables y_i is unduly large, where $0 < i < \lceil \log_2 s \rceil - 1$ (s is the number of states of the automaton. In Sec. 7, we show how to overcome the first of these drawbacks and in Sec. 6, how we can surmount both of them.

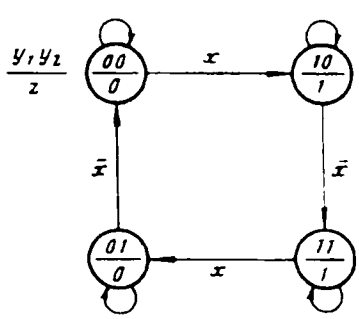

Fig. 15

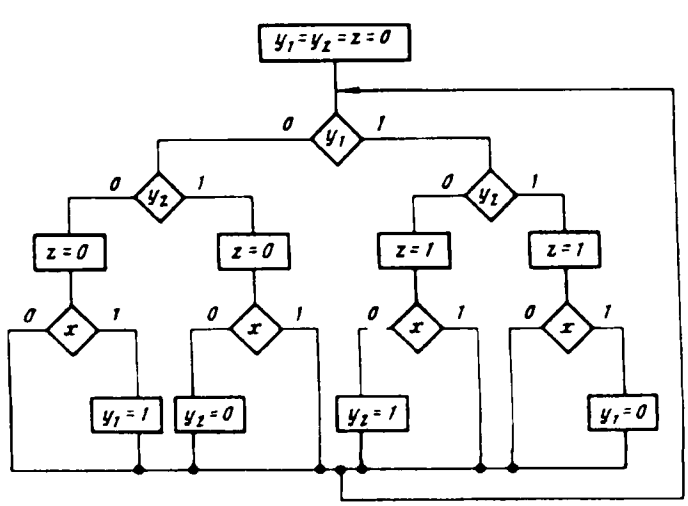

Fig. 16

6. MOORE MACHINES WITH BINARY (UNARY) STATE CODING

Owing to such a coding of the vertices in Moore machine transition graphs (Fig. 17), which is referred to as unary coding in [2], (the binary variable y_j takes the value one only at the jth vertex, and zero at all other vertices), we can use a linear state decoder (Fig. 18) instead of a pyramidal decoder, which is used in Moore machine transition graphs for other coding variants. The drawback of the algorithmic graph scheme in this case is that the number of additionally introduced variables y_j $(0 < j < s - 1)$ is far larger than in the previous case. Furthermore, each of these variables is to be defined or discarded forcibly.

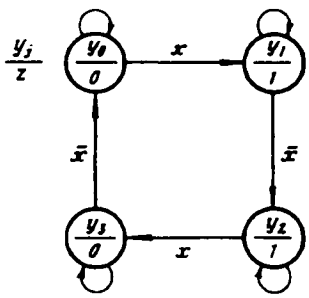

Fig. 17

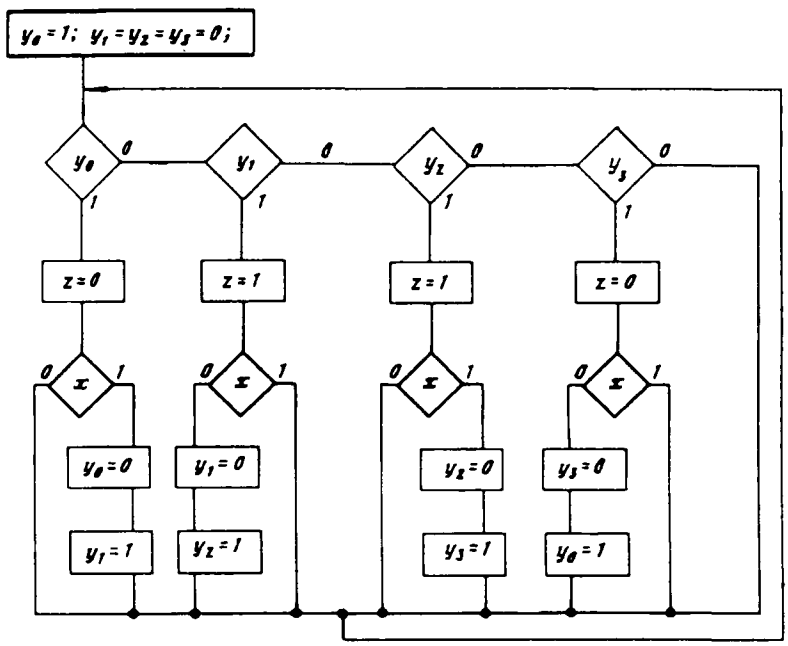

Fig. 18

7. MOORE MACHINES WITH MULTIVALUED STATE CODING

The disadvantages inherent in the previous coding methods are eliminated by using multivalued coding of the states of an automaton [5, 6]. For the Moore, Mealy, and combined-type automata implemented as one unit, their states can be coded by one s-valued variable Y, which needs no forcible elimination since it is automatically reset when passing from one value to another. For implementing an N-component control algorithm (realized by N graphs) by these classes of automata A, we require N multivalued variables Y_j, where $j = 0 \ldots N - 1$. We believe that this coding variant, which is almost not realizable in modern automata, will become the basic method of software realizations of automata when no stringent constraints are imposed on the memory space occupied by programs and when there is the possibility of using multivalued variables.

Figure 19 shows an algorithmic graph scheme in this coding variant constructed from the Moore machine transition graph of a trigger (Fig. 7 in [1]). In this graph scheme, the conditional vertex $Y = 3$ and the second operator vertex $z = 1$ can be discarded by slightly reducing the degree of understandability.

An algorithmic graph scheme of such similar structure can be constructed for any Moore machine transition graph. However, unlike graph schemes, such an algorithmic graph scheme will always be planar. The application of only one intermediary variable, if the structural organization is "good," will make an algorithmic graph scheme of this type attractive in practice. The only disadvantage of this type of structures, besides being cumbersome (when the values of output variables are not omitted) is that the generating circuits are not easily detected by inspection, since the fragments are not directly connected as in any transition graph, but are related indirectly through the values of the variable Y.

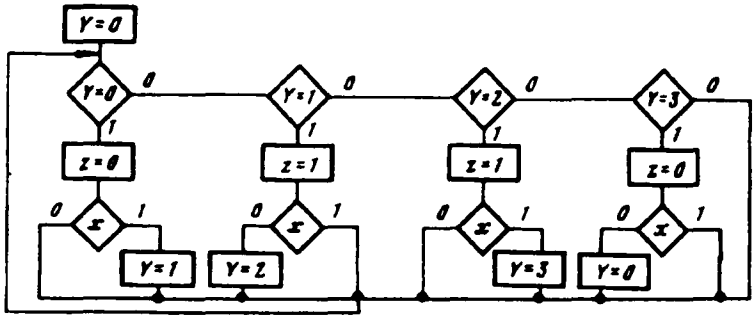

Fig. 19

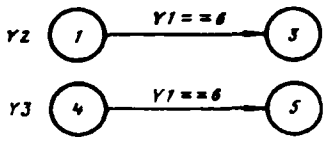

Fig. 20

Hence, in those cases in which an algorithmic graph scheme without internal feedback is chosen as a communicative language for some specific reason, it is more preferable to use algorithmic graph schemes with a decoder for decrypting multivalued state codes.

Multivalued coding additionally opens a way (effective from the viewpoint of graphic representation) to providing connections between control algorithm components (even in parallel processes), because the decimal values used in the interaction of the numbers of states can be substituted for the components (interaction with input, output, and additionally introduced internal variables is preserved). For example, let us assume that the second component is to be transformed from the first state into the third state, and the third component is to be transformed from the fourth state into the fifth under the condition that the first component exists in the sixth state. Without introducing any additional variables, this parallel process can be visually represented by a fragment of the system of interconnected transition graphs shown in Fig. 20. Interconnected transition graphs, as demonstrated in [7], possess wide graphic representation capabilities; in particular, they depict such important properties as parallelism and hierarchism in explicit form.

However, a transition that is sequential with respect to states is usually a parallel process with respect to inputs and outputs (see the transition graphs shown in Figs. 7 and 8).

8. MEALY MACHINES WITH MULTIVALUED STATE CODING

Mealy machines, like Moore machines, admit the use of various types of state coding. But, in light of what has been discussed above, we only examine multivalued coding. Figure 8 in [1] shows a Mealy machine transition graph capable of implementing a flip-flop counter for multivalued coding, and Fig. 21 shows the corresponding algorithmic graph scheme. The conditional vertex $Y = 3$ and the secondary operator vertices $z = 1$ and $z = 0$ can be excluded from this graph scheme. This graph scheme differs from the Moore machine graph scheme in the location of the layer containing operator vertices labeled by the values of the output variable: this layer is not located before the decoders which decrypt the values of input variable, but after them.

Another example of implementing this class of automata is shown in Fig. 22 for a sequential single-digit summator (Fig. 10 in [1]). In this graph scheme, the number of vertices is minimized by placing the layer containing the values of the output variable z after the layer containing the values of the succeeding state p. This graph scheme, containing in all nine vertices, is constructed by a modified Bloch method [8].

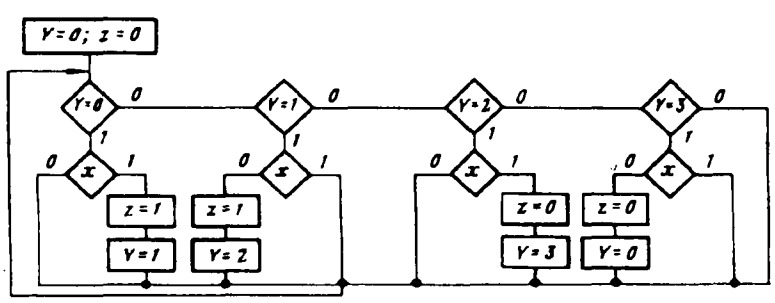

Fig. 21

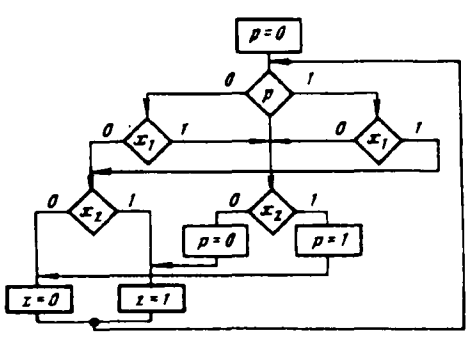

Fig. 22

Figure 23 shows an algorithmic graph scheme implementing a combined-type C-automaton with flag (see Fig. 11 in [1]). This graph scheme contains five layers: a state decoder, a layer for forming the values of output variables, a layer for implementing transition functions, a layer for forming the flag values, and a layer for forming the succeeding state. The variable x_1 in this scheme determines the contents of the flip-flop counter (another component of the control algorithm), which is also controlled by other data sources—buttons. Unlike in transition graphs, contradiction in this scheme is eliminated by orthogonalization and use of the Boolean criterion T instead of the inequality $t \geqslant D$.

While every component is closed in combined-type transition graphs, contact operation in graph schemes incorporated in programmable logical controllers [9] is effected for the control algorithm as a whole.

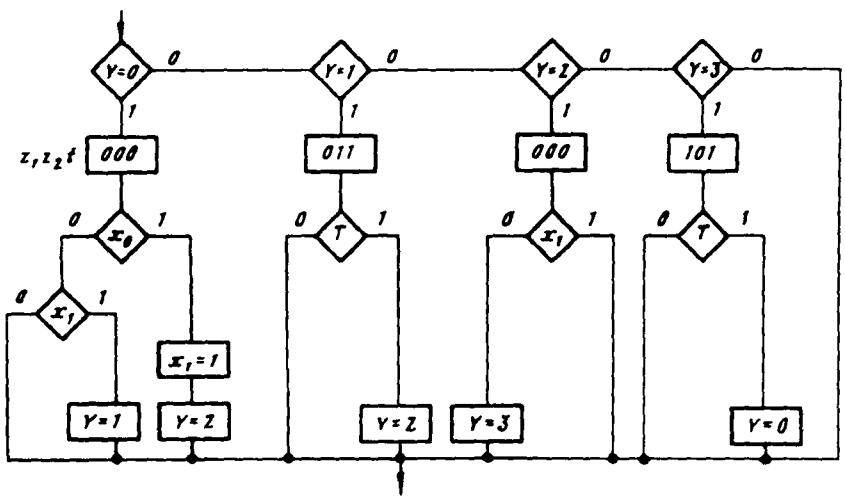

Fig. 23

10. PROGRAMMING OF TRANSITION GRAPHS AND ALGORITHMIC GRAPH SCHEMES WITH MULTIVALUED STATE CODING IN HIGH-LEVEL LANGUAGES

Using multivalued coding, we can convert AGS4 schemes and graph schemes isomorphically into program codes without constructing intermediary graph schemes [1]. Although graph schemes contain loops and (non-generating) circuits, there is no need to deloop or structurize them. These problems can be solved in the course of programming through appropriate choice of control constructs.

The easiest way of achieving this end is to use such a controlling construction like the switch operator in the C language [5, 6, 10]. Let us illustrate the use of this switch for realizing a Moore machine transition graph (Fig. 7 in [1]) or an algorithmic graph scheme (Fig. 19).

```
                switch (Y)      {
case 0 :        z = 0;
                if (x)          Y = 1;
                break;
case 1 :        z = 1;
                if (!x)         Y = 2;
                break;
case 2 :    /* z = 1; */
                if (x)          Y = 3;
                break;
case 3 :        z = 0;
                if (!x)         Y = 0;
                break; }
```

This program has a fascinating property—along with full guarantee, it does only what is described in the transition graph and does not perform anything else. This can be verified by collating the program code with the transition graph, because complete isomorphism must exist between them when the transition graph is transformed into the program code without any error. This property is not inherent in all algorithmic models. For example, if an automaton A with s states is realized by a system of m Boolean formulas $(m = \lceil \log_2 s \rceil, s \neq 2^m$, then the system realizes another automaton with 2^m states, in which the initial automaton only serves as a kernel for the newly constructed automaton, and transitions from the new $2^m - s$ states into given states will depend on the method applied to complete the definition.

In this program, the value of the output variable that does not vary during transition from the preceding state is deleted from state 2 (case 2). On the one hand, this reduces the memory space occupied by the program, and, on the other hand, preserves good readability of the program. Every BREAK operator in this program transfers controls outside the closing brace. Consequently, if the condition the IF operator is not satisfied, the preceding state is retained. But, if the condition is satisfied, not more than one transition is accomplished in the transition graph. However, not more one transition per program cycle is accomplished in every transition graph of combined-type transition graphs. Hence the number of any state of the automaton is accessible to the other components of the control algorithm, no matter what the values of the input variables. On the contrary, in AGS2 (Fig. 3 in [1]), the value of $z = 1$ for $x_1 = x_2 = 1$ is not accessible to other control components.

For example, if the first component of the control algorithm is realized as described above and the value of the variable $Y1$, which is equal to six, is accessible to the other components, then a fragment of the interconnected transition graph (Fig. 20) corresponds to the succeeding fragment of the program which uses the sixth state of the first component as a transition condition in the other two components:

```
                switch (Y2)        {
                ...
case 1 :        if (Y1 == 6)                Y2 = 3;
                ...                }
                switch (Y3)        {
                ...
case 4 :        if (Y1 == 6)                Y3 = 5;
                ...                }.
```

From the foregoing, we discover a very unpleasant fact—the specific properties of software realization distort not only the reading of a program code, but also the reading of specifications of transition graphs or combined-type transition graphs. Therefore, to ensure universality of graphs, including transition graphs with unstable vertices, we must preferably assume that not more than one transition is accomplished in passing through every transition graph with proper program support.

We now give an example of a program constructed from a Mealy machine transition graph (Fig. 8 in [1]) and an AGS4 (Fig. 21), assuming that initially $Y = 0$ and $z = 0$:

```
                switch (Y)         {
case 0 :        if (x)             {z = 1;   Y = 1;}
                break;
case 1 :        if (!x)            {/*z = 1; */Y = 2;}
                break;
case 2 :        if (x)             {z = 0;   Y = 3;}
                break;
case 3 :        if (!x)            {/*z = 0; */Y = 0;}
                break;             }.
```

Using SWITCH control constructs, we can effectively implement C-automata as well.

First, a transition graph can be used as a specification for control algorithms, as well as programs. Second, if SWITCH control constructs are incorporated, this graph can be isomorphically mapped into the code of a structured program, which can be observed not only by binary outputs, but also, and this is most important, by the decimal numbers of states. Moreover, it suffices to display only one decimal internal variable (signature) for every transition graph of Moore machine or Mealy or combined-type automaton on the monitor in order to verify the program. This will enable us to study the dynamic behavior of an automaton by tracking the values of this variable (if, for example, for a Moore machine, the correspondence between the number of the state and the values of output variables in this

state is preliminarily determined), instead of using several binary internal variables as practiced in the traditional approach.

Thus, in programming, we can introduce the concept of "observability" in analogy with the Zadhe concept of "measurability" used in automatic control theory [11].

The programs thus designed exhibit high capabilities (in the sense of easy control) and are easily readable, provided the values of output variables are defined uniquely, because they, in this case, depend on the prehistory. Transition graphs of automata with flags can also be mapped isomorphically into program codes with the help of SWITCH control constructs. But, since the succeeding state depends on the prehistory, they perceptibly lose their modification capabilities.

The main reason why changes can be easily introduced in programs thus constructed from transition graphs is that the graph vertices are directly interconnected. On the contrary, in graph schemes in which operator vertices are largely joined through conditional vertices, the changes taking place in a transition generally exert considerable influence on other transitions.

In our approach, a program can be easily verified by collating the program code with its transition graph. For one-component control algorithms, the transition graph can be used as a test for verifying the program, whereas for multi-component control algorithms (in analogy with Petri networks), we must construct, using transition graphs of combined-type automata, a reachable label graph that is helpful in studying the functional capabilities of the system and represent the behavior of the system by one transition graph. This problem is resolvable in the absence of any parallelism between the states in control algorithms or for low-dimensional problems. There is no need to construct a common transition graph for independent parallel processes, because each of these processes can be verified separately.

11. PROGRAMMING OF ALGORITHMIC GRAPH SCHEMES WITH INTERNAL FEEDBACK IN HIGH-LEVEL LANGUAGES

The approach designed in this paper is useful in constructing a direct programming technique for algorithmic graph schemes with internal feedback. Such schemes in multi-task mode cannot be realized by the traditional method without preliminary delooping [1].

The method essentially consists in constructing from a given algorithmic graph scheme an equivalent transition scheme, which is isomorphically mapped into the program code with a SWITCH control construction.

Let us suppose that we are given an algorithmic graph scheme with internal feedback (Fig. 1). Using this scheme, we must construct a Mealy (Fig. 2) or Moore (Fig. 3) machine transition graph in such a way that each machine is uniquely realized by a SWITCH construction. Although the programs thus designed are compact, their transition graphs contain generating circuits which are not easy to understand because of the ambiguity in the values of the output variables. Programs constructed from a transformed Moore machine transition graph (Fig. 7) do not suffer from such shortcomings:

```
        switch (Y)          {
case 0 : z1 = 0;            z2 = 0;        z3 = 0;
         if (x1&x3)                        Y = 1;
         if (x1&x2&x3)                     Y   2;
         break;
case 1 : z1 = 1;            /* z2 = 0;    z3 = 0; */
         if (!x3)                          Y = 0;
         if (x3)                           Y = 1; break;
case 2 : /* z1 = 0; */          z2 = 1;    /* z3 = 0; */
         if (!x3)                          Y = 0;
         if (x3)                           Y = 4; break;
case 3 : /* z1 = 0; */          z2 = 0;    /* z3 = 1; */
                                           Y = 0; break;
case 4 : /* z1 = 0;            z2 = 1;    */ z3 = 1;
         if (!x3)                          Y = 3;
         break;
case 5 : /* z1 = 1;            z2 = 0;    */ z3 = 1;
         if (!x3)                          Y = 0;
         break;                }
```

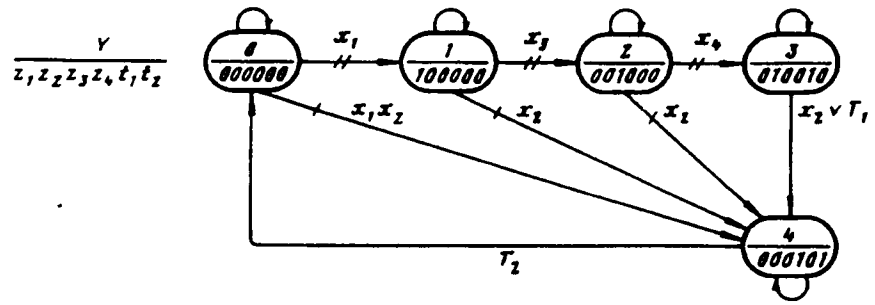

Fig. 24

The number of rows in this program is $s + d + 1$, where d is the number of arcs (including loops) in the transition graph. In CASE 0, the IF operator corresponding to the arc with the highest priority emerging from the vertex 0 is located below the IF operator corresponding to the arc with the lowest priority. In such a software realization, the problems of delooping and structurization of the initial algorithmic graph scheme are automatically solved in the course of designing the program. It should be mentioned that here the construction of a transition graph from an algorithmic graph scheme with internal feedback is not obligatory. To write a program in this case, it suffices to apply multivalued coding (Sec. 2) for the operator vertices in the algorithmic graph scheme (in order to construct a program corresponding to the Moore machine) or for the points located after the operator vertices (in order to construct a program corresponding to the Mealy machine).

The speed of a program can be enhanced by refining its structure:

```
                switch (Y)          {
case 0 :        z1 = 0;             z2 = 0;       z3 = 0;
                if (x1&x2&x3)                     {Y = 2; break;}
                if (x1&x3)                        Y = 1;
                break;
case 1 :        z1 = 1;             /* z2 = 0;    z3 = 0; */
                if (!x3)                          Y = 0;
                if (x3)                           Y = 1; break;
case 2 :        /* z1 = 0; */       z2 = 1;       /* z3 = 0; */
                if (!x3)                          Y = 0;
                if (x3)                           Y = 4; break;
case 3 :        /* z1 = 0; */       z2 = 0;       /* z3 = 1; */
                                                  Y = 0; break;
case 4 :        /* z1 = 0;         z2 = 1;       */ z3 = 1;
                if (!x3)                          Y = 3;
                break;
case 5 :        /* z1 = 1;         z2 = 0;       */ z3 = 1;
                if (!x3)                          Y = 0;
                break;              }
```

In this program, the fragment which corresponds to an arc of high priority emerging from the zero vertex is located above the fragment corresponding to the arc of lower priority emerging from the same vertex.

By way of another example, let us construct the equivalent Moore machine transition graph (Fig. 24) for an algorithmic graph scheme with internal feedback (Fig. 2 in [1]). In this transition graph, which is a component of the controller, to every unit at the positions t_1 and t_2 there is an inversion of TIME operation (i, D), where D is the delay of the ith delay unit. This transition graph, like the previous graph, is implemented with a SWITCH construction.

12. PROGRAMMING OF ALGORITHMIC GRAPH SCHEMES AND TRANSITION GRAPHS WITH MULTIVALUED STATE CODING IN LOW-LEVEL LANGUAGES

As observed in [1], in order to take account of the properties of control constructs of the programming languages used, we must preferably transform the initial algorithmic graph scheme into program codes not directly, but through an intermediary graph scheme (which is referred to in [1] as AGS3).

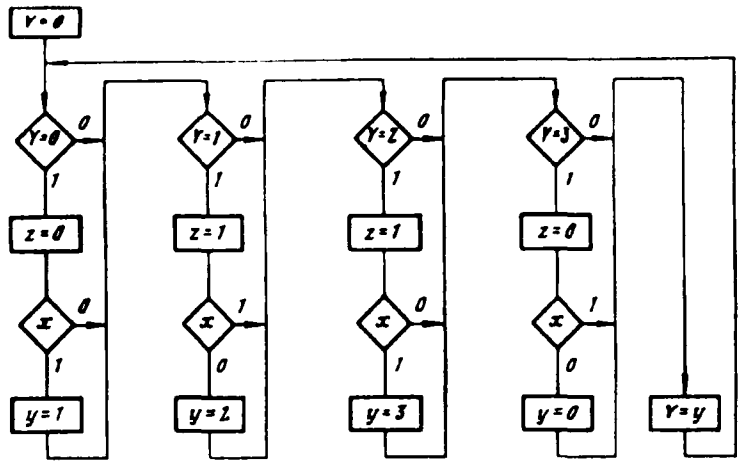

Fig. 25

Figure 25 shows an algorithmic graph scheme which is the equivalent of the algorithmic graph scheme shown in Fig. 19 and which is linearized and structurized in a special manner (every conditional vertex in every block transfers the control to one point if the conditions are not satisfied. This equivalent graph scheme, in turn, can be almost isomorphically transformed into an algorithmic graph scheme which takes account of the semantics of the commands and the computer architectural specifics, for example, programmable logical controller [12], by which the program code can also be isomorphically constructed in the mnemonic code of the controller:

```
STR   R   C  0; Input 0 in registering summator (RS:=0)
EQU   R   M  Y; if (Y = 0) binary summator (BS=1)
IF        T   ; if (BS) go to the next command, else to
                ; label CONT
EQ        RO  z; if (BS) z = 0
IF          I x; if (x) go to next command; else to
                ; label CONT
STR   R   C  1; RS:= 1
EQ    R SM   y; if (BS) y = 0
CONT        .   ; go to next command

STR   R   C  1
EQU   R   M  Y
IF        T
EQ        SO  z
IF          I  x
STR   R   C  2
EQ    R SM   y
...

STR   R   M  y; RS := y
EQ    R   M  Y; Y := RS
STOP            ; transfer control to program start
```

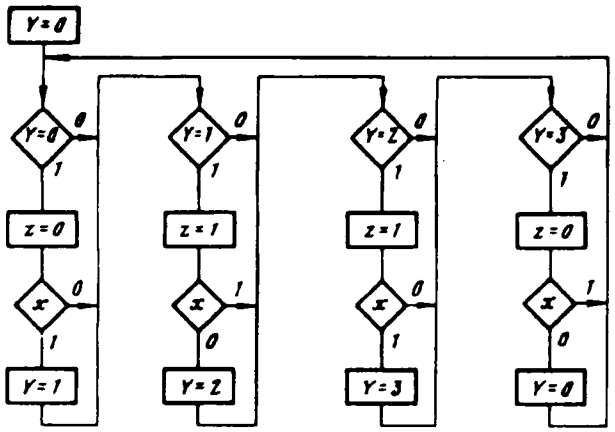

Fig. 26

Instead of an algorithmic graph scheme, using a transition graph (Fig. 17 in [1]) as a specification language, we can almost isomorphically transform the transition graph into the program code

```
STR   R   C  0;  Input 0 into registering summator (RS:=0)
EQU   R   M  Y;  if (Y = 0) binary summator (BS)=1
EQ        RO z;  if (BS) z = 0
AND        1 x;  BS:=BS&x
STR   R   C  1;  RS:=1
EQ    R  SM  y;  if (BS) y = 1

STR   R   C  1;  RS:=1
EQU   R   M  Y;  if (Y = 1) BS=1
EQ        SO z;  if (BS) z = 1
AND        1 x;  BS:=BS&x
STR   R   C  2;  RS:=2
EQ    R  SM  y;  if (BS) y = 2
...
STR   R   M  y;  RS := y
EQ    R   M  Y;  Y := RS
STOP             ; transfer control to program start
```

This program contains fewer commands than the previous program. It does not include any conditional transitions and all commands in it are executed sequentially. Therefore, the program can be further simplified by excluding the third command, provided the ninth command is replaced by the command EQ 0 z (z := BS) under the condition that z is equal to 0 at the program start. This is explained as follows: as long as automaton A exists in the first vertex, a unit value is entered in the cell z and when A "quits" this vertex, a zero value is entered. The program can be simplified still further by reducing the degree of its isomorphism with the transition graph. Furthermore, each pair of commands 5–6 and 11–12 can be replaced by an INC R M y ($y := y + 1$) command.

The programs examined above can execute one program cycle of not more than one block in the algorithmic graph scheme (Fig. 19) or one transition in the transition graph (Fig. 7 in [1]), since an auxiliary multivalued internal variable y is incorporated for holding the value of Y in the course of a program cycle.

However, it must be mentioned that one auxiliary variable y is adequate for realizing any system of interconnect transition graphs. Discarding the condition that not more than one transition in the transition graph is accomplished in one cycle, we can transform the algorithmic graph scheme of Fig. 25 into the algorithmic graph scheme of Fig. 26. Thus we can greatly simplify the programs presented above. The algorithmic graph scheme thus constructed is valid, because the input variables do not change their values in the programmable logical controller [12] when the program executes a cycle.

If a transition graph is used as a specification language and the constraint restricting the number of transitions executed in one program cycle is discarded, then such unconventional constructs like step register (SR) inherent in the instruction sets of many programmable logical controllers [9, 12] can be used for compiling programs. In [9, 12], a step register is used for generating sequential steps and no mention is made of its use in implementing arbitrary automata. Therefore, we believe that our studies pave the way for such a use.

By way of example, we present a program implementing a transition graph (Fig. 7 in [1]), in which the zero (out of 32) step register initially exists in the zero step (the zero state) (out of the 256 permissible steps):

```
READ    S  0; Selection of a step register as 0 step
STR     S  0; if (S=0) BS=1
EQ     RO  z; if (BS) z = 0
AND     I  x; BS := BS&x
STEP    S  1; if (BS) S=1
STR     S  1; if (S=1) BS=1
EQ     SO  z; if (BS) z = 1
AND    NI  x; BS := BS&!x
STEP    S  2; if (BS) S=2
...
STOP
```

Since this program also does not contain any conditional transitions, the third command can be discarded by replacing the seventh command by the command EQ 0 z, provided $z = 0$ at the initial instant.

The approaches presented in this section are equally applicable to Mealy machines, combined-type C-automata, and automata with flags. For example, the Mealy machine transition graph (Fig. 8 in [1]) describing the program below written in C language

```
    Y = 0;
M : if((Y == 0) & x)    {z = 0;  Y = 1;}
    if((Y == 1) & !x)   {z = 1;  Y = 2;}
    if((Y == 2) & x)    {z = 1;  Y = 3;}
    if((Y == 3) & !x)   {z = 0;  Y = 0;}
    goto M;
```

is implemented by the following program incorporating step registers:

```
READ    S  0  STR     S  1
              AND    NI  x
STR     S  0  EQ     S0  z
AND     I  x  STEP    S  2
EQ     RO  z  ...
STEP    S  1  STOP
```

From the foregoing, it is clear that the step registers aid in greatly enhancing the level of mnemonic programming of automata. This is also observed in implementing controllers, because in many cases delay units can oe realized through the NEXT Si D command, which, if a step register resides for D seconds at the ith step, will promote transition to the $i+1$th step. In order to apply this command, the initial transition graph may have to be appropriately transformed (by increasing the number of vertices) so that transition is effected only to the adjacent vertex at the end of the time delay. However, in this case, the controller is not decomposed into an automaton and delay units, but is programmed as a single entity. By way of example, we now construct a Moore machine transition graph with flag (Fig. 11 in [1]), preliminarily increasing the number of vertices in the graph to five, an assuming that $z_1 = z_2 = 0$ at the start of the program:

```
READ    S  0  STR     S  2
IF      S  0  AND    NM  x1
STR     M  x1 STEP    S  3
STEP    S  1
STR     I  x0 STR     S  3
EQ     SM  x1 EQ      0  z1
STEP    S  2  NEXT   S3  D
CONT
                STR     S  4
STR     S  1  STEP    S  0
EQ      0  z2
NEXT   S1  D  STOP
```

Isomorphism of this program with the transition graph is guaranteed by the vertices in the transition graph. Here the program fragment corresponding to the arc with higher priority emerging from the zero vertex is located below the fragment corresponding to the arc of lower priority emerging from the same vertex. Since the commands which transfer the output values of output variables are contained only in the "linear" segment of the program (i.e., not "protected" by an IF command), no commands are used to form the zero values of output variables, and the commands EQ 0 z_i are used to generate their unit values.

If isomorphism is ensured not only with respect to the vertices, but also with respect to the arcs in the transition graph, then the program fragment corresponding to an arc of higher priority is located above the fragment corresponding to an arc of lower priority:

```
READ    S  0     STR     S  2
                 AND     NM x1
STR     S  0     STEP    S  3
AND     I  x0
EQ      SM x1    STR     S  3
STEP    S  2     EQ      0  z1
STR     S  0     NEXT    S3 D
AND     M  x1
STEP    S  1     STR     S  4
                 STEP    S  0
STR     S  1
EQ      0  z2    STOP
NEXT    S1 D
```

13. OUR APPROACH VERSUS THE ASHCROFT-MANN METHOD OF CONSTRUCTING STRUCTURED GRAPH SCHEMES

Ashcroft and Mann [13, 14] designed a universal method of constructing structured algorithmic graph schemes. Using their method, we can construct a structured AGS1 or AGS2 with multivalued state coding from unstructured AGS1 or AGS2 (see [1] for the definitions of AGS1, AGS2, AGS3, AGS4, and AGS5).

In our opinion, their method is not helpful in constructing understandable algorithmic graph schemes because of the following deficiencies:

(1) the concept of a "state" is not used in explicit form,

(2) automata and variables are distinguished by types,

(3) only one type of coding is used for the fragments of an unstructured algorithmic graph scheme in combining them into a canonical structure,

(4) the generating "circuits" are visually not detectable from data due to the interconnections between the fragments of a structured algorithmic graph scheme,

(5) the depth of a structured algorithmic graph scheme is not restricted to one transition, and this may hinder the use of the values of certain inte..al variables in other co..ponents of the control algorithm,

(6) the ambiguity in the output values is not eliminated when AGS1 is used as the initial representation,

(7) when an AGS2 is used as the initial representation, the ambiguity in the output values is not eliminated and, additionally, AGS2 may contain a large number of binary intermediary and flag variables (even omitted values) which do vanish in the course of structurization and an additional multivalued variable corresponding to the numbers of selected structured fragments is introduced,

(8) man gets accustomed to and understands the familiar initial representation of the control algorithm, and is psychologically not prepared to handle other novel forms of algorithmic representations, and

(9) the method is primarily oriented for use with high-level languages.

Since their method is designed for developing structured algorithmic graph schemes, as implied in its design philosophy, it must not be applied to an already structured algorithmic graph scheme. For this reason, if a structured AGS2 is used as the initial representation (Fig. 3 in [1]), from what has been said above, it follows that the initial representation must not be subjected to further transformation. But, as demonstrated in [1], this algorithmic graph scheme is "badly" understood and, therefore, the algorithmic graph scheme shown in Fig. 10 or the transition graph shown in Fig. 11 is preferable to a structured AGS2 as a means of communication.

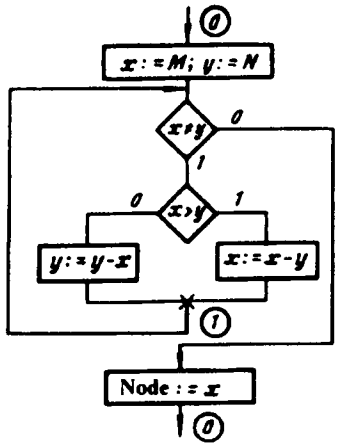

Fig. 27

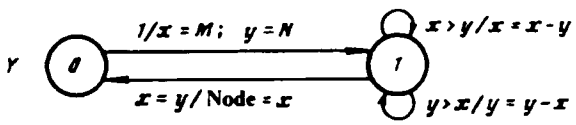

Fig. 28

Thus, the idea due to Ashcroft and Mann is full of defects, because it is not clear whether or not to construct first an unstructured algorithmic graph scheme and structure it subsequently. What is most important is that right from the start we must either straightforwardly construct a structured algorithmic graph scheme with a state decoder for the coding variant used, or, what is more preferable, implement the initial description as a transition graph for the chosen type of automata, which (whenever possible) must not contain any omitted values of variables and which is isomorphically implemented, say, by SWITCH constructs of the C language capable of concurrently executing delooping and structurizing operations and at the same time providing access to any value of the multivalued variables representing the states of the automaton. The approach is also applicable to low-level languages, for example, mnemonic codes of programmable logical controllers.

If the concept of a "state" is introduced, then every component of the program becomes intrinsically "observable" (through a multivalued variable) with respect to not only inputs, but also outputs. Moreover, it is a simple matter to introduce changes in the programs thus compiled. Owing to the isomorphism of these programs with the initial description, they easily yield to verification (when the transition graph is used as a test).

Transition graphs, by form of representation, being in general "essentially planar," transform control algorithms in a more natural form than a graph scheme or Grafset diagram [9] that are transformed, in compliance with standards, usually in a form close to top-to-bottom linear representation. Such a representation is in agreement with book layout and general reading practice, but not with planar (parallel) picture representation, which is more convenient for man's perception. Naturally, transition graphs must be as planar as possible. Unlike the other graphs examined in this paper, transition graphs form the initial "framework" of algorithmization theory. Therefore, according to the Occam principle [4], which asserts that "essence must not be overemphasized unless there is a necessity," such a necessity hardly arises in most logical control problems.

Nevertheless, transition graphs, unlike transition tables based on minterms and usually containing a large number of empty cells, are tangibly more visible and virtually applicable to every large-dimensional problem. The main advantage of transition graphs in describing large-dimensional problems is that the transition between two vertices in an orthogonalized graph is determined by not all input variables stated in the labels of all arcs as in transition tables, but only by those input variables that label the arc between a given pair of vertices. This property is known as local description.

If these contradictions are eliminated not by orthogonalization, but by proper placement of the priorities, omissions of certain input variables worsen the readability of the transition graph, because it is not possible to determine (read) at once the list of all variables governing every transition from arcs with lower priorities emerging from a vertex. To find this list of variables, we must examine the labels of all arcs emerging from a given vertex. But the locality of description in this case is reduced.

The hurdles encountered in large-dimensional problems when using a transition table as the main model for which algorithmization algorithms are available for use in binary apparatus, especially in asynchronous implementation, constitute, probably, the restraining factor, which, by tradition, is also extended to transition diagrams (the basic name used in algorithmization theory to denote transition graphs). These diagrams, which are used as an auxiliary (illustrative) model in algorithmization theory, thus far hinder the extensive use of transition graphs as a communicative and specification language in software realization of logical control problems for which the traditional problems of the algorithmization theory are either nonexistent or not decisive.

In my opinion, in light of the present-day developmental level of the elemental base of software realization of practical logical control problems, the fundamental criterion for program design must be good readability and,

as a consequence, buglessness. All other criteria must merely serve as ancillary tests (although limited resources of programmable logical controllers may dictate the choice of models for algorithmization). Therefore, the ideas developed in this paper may be expected to promote the use of transition graphs in software realization of this class of problems.

Our method is useful not only in logical-time control algorithms, but also in logical-computing algorithms. Figure 27 illustrates an AGS1 implementing an algorithm for determining the greatest common divisor of two positive integers M and N (the Euclidean algorithm). This algorithmic graph scheme is not an automatic, but a logical-computing scheme. Being a flagged Mealy machine, this scheme can be programmed by our method from a transition graph describing the logical-computing process (Fig. 28). It must be noted that this transition graph is far more compact than the corresponding algorithmic graph scheme.

Finally, I hope that my method may prove helpful to object-oriented programmers as a mathematical tool for handling the "states" introduced for describing "objects."

14. CONCLUSION

The method presented in this paper, which we refer to as the SWITCH-technology of algorithmization and programming of logical control problems, is one of the rapidly advancing recent trends in CASE-technology [15] (Computer Aided Software Engineering) (surprising coincidence between the prefixes CASE and SWITCH). There is "virtually no alternative" to this technology, "because it is absolutely impossible to compile specifications for a system that would adequately take account of all finer details of the behavior without special methodology and apparatus" [16]. This is especially true in light of one of Murphy's laws: if something is apparently simple, it is usually complicated; if something is apparently complicated, it may generally be unrealizable [4].

Our approach gives all developers not only a chance to control the behavior of the "black box"—a program—as is usually done, but also permits them to "peep into" the box in order to observe the internal states of each of the components, by tracking only one component, and to verify, if the need arises, the program codes implementing the functional tasks. For this purpose, I elaborated a program shell for configuring large-dimensional systems of control algorithms defined by interconnected transition graphs and written in C, and jointly with Kuznetsov (Avrora, Inc.) developed a compiler for the C-instruction set incorporating the specifics of software realization of automata in programmable controllers described in [12].

The program shell simulates not only control algorithms, but also control algorithms along with the controlled object as an integrated system, in which the components of the model of the object (for example, valves) are also described by transition graphs. Furthermore, the shell provides an opportunity to change the values of input variables and to observe the generated values of all output and time variables, as well as the values of the numbers of the states of every component of the system both in step-by-step (one program cycle) and in automatic (from one stable state to another) operation modes.

Our approach clearly partitions the tasks and, most importantly, the responsibilities between customer (technologist), developer, and programmer in the event that they represent different institutions, especially different countries; otherwise, language and, eventually, economic barriers would arise. In this approach, the programmer need not necessarily know the specifics of the technological process, and the customer is not assumed to possess programming skills. Moreover, the approach paves the way for communication between the customer, developer, programmer, and operator not through the traditional route in terms of the technological process (for example, emergency start is "ON"), but through a completely formalized language (a specific sort of technical esperanto). For example, the progress message, "in the third transition graph of the fourth unit, the value at the fifth vertex has changed from 0 to 1" will not be interpreted by different people differently, as may happen in any language (for example, "ambassador lies abroad for the good of his country") [17], and does not require the service of specialists knowledgeable in technological process in order to make changes. Furthermore, if programming is left to the "discretion" of the developer (of course, this may not be always true of developers working with certain foreign companies), he has a right and may refuse to seek the services of a functional programmer and take recourse to computer-aided programming.

Nevertheless, in most cases, while using algorithmic graph schemes, it is not a simple matter to switch over from algorithmization to programming in problems of logical control of complex technological processes. This can be explained as follows: almost always algorithmization does not end there where it ought to end—the creation of an algorithm in the mathematical sense, which, by definition, must uniquely accomplish every computation and terminate in some "pattern" called the algorithm, which is to be conceived to some degree in the course of programming. In this situation, either the developer himself must compile the program, or the programmer must be

aware of the minutest details of the technological process, or both of them together must eliminate the inevitable pitfalls inherent in the traditional design of programs via testing.

In our approach, algorithmization must proceed and end in a different level--in the process of interaction between the customer and developer. Delivery of technical specifications is a single one-time event followed by subsequent addenda and ends with the creation of a system of interconnected transition graphs, which takes full account of the minutest details of every transition. Here there is no need for the programmer to invent anything for the functional problems; he is merely called upon to realize this system of transition graphs in a unique way. Consequently, the stringent requirements on the programmer's skill and qualification can be slackened.

Our approach was successfully tested by AVRORA, Inc. (Russia) jointly with NORCONTROL Ltd. (Norway) in designing a logical control system for a marine diesel generator [18] and in creating a control system for the same type of diesel generator, "SELMA-2," manufactured by ABB STROMBERG (Finland) [19]. In the first case, programming was implemented in the PL/M high-level language, whereas in the second case, programming was implemented in a special functional block language.

For the second case, jointly with V. N. Kondrat'ev (Avrora, Inc.) we developed a scheme based only on the use of functional library blocks (digital and logical multiplexers) which ensure isomorphism between the transition graphs approved by the customer and the generated functional scheme. This approach radically differs from the traditional method which aims at achieving only functional equivalence between the functional scheme and specifications (if any), but does not ensure their selective equivalence—a useful tool for verifying the image representations. In the traditional approach, a functional scheme realizes a given behavior, but describes this behavior in a form that does not reveal the dynamics of transitions from one state into another or the dynamics of the changes taking place in the values of output variables of the automaton.

While the specifications realized by control algorithm are implemented with the help of a functional scheme based on triggers and logical circuits incorporating feedbacks, it is rather difficult to understand a functional scheme, although it is, unlike the original specifications, completely defined (completeness (if any) is lost in the course of construction of the scheme) and does not contain omissions (except for, probably, the types of triggers not specified in the scheme); therefore, like transition graphs, it can be formally and independently programmed. The reason for this situation is that, first, a functional scheme, in contrast to transition graphs, is not local in description (if the scheme contains several input variables, only an in-depth analysis can reveal which of these variable exercises influence on a transition from a given state into the succeeding state). Second, in interconnected realization, a functional scheme does not exhibit the property of locality with respect to changes and, this is most important, does not contain any predefined values for states and output variables. Therefore, reading a scheme consists in logically computing the values of intermediary variables of triggers and in storing these values at the "head" of the scheme. A functional scheme is usually tested (for consistency, missing generation, sequence of generation of the values of output values, etc.) by feeding the input action and then computing the output responses. It is not a simple matter to design these tests, because triggers and feedbacks create serious difficulties in designing suitable tests for determining the functional capabilities of a scheme. Therefore, verification is preferable to testing, i.e., formal compilation of a system of Boolean formulas for the functional scheme, formal construction of a transition graph on the basis of this system, and an analysis of the behavior of the transition graph.

If the transition graph thus constructed behaves differently than what it is expected to do, then, instead of changing the initial functional scheme, we must construct a corrected transition graph, which, in turn, can be realized through various algorithmic models, even models differing from the functional scheme. In using the functional scheme along with the new transition graph, we must formally construct a new system of Boolean formulas such that it can be realized formally by the functional scheme.

Although in hardware (especially in asynchronous) realization, a real scheme may exhibit behavior different from the behavior defined by the model, in software realization this discrepancy is easy to avoid; for example, in a system of Boolean formulas, this hurdle can be surmounted through the use of a different set of notations for the internal variables.

An analogous approach can be applied when relay-controlled (ladder) circuits are used as the programming language.

Our approach aids in programming from a unified standpoint in different languages used in modern programmable logical controllers like the programmable logical controllers [20] for which programming can be carried out, in languages recommended by the international standard IEC 1131 (sequential function chart, function block diagram, ladder diagram, and instruction list languages), as well as in Assembler and C languages.

Our approach is idealogically close to the method used in elaborating the "YARUS" language by Kuznetsov (Institute of Control Sciences, Moscow) [21] and its modifications [22], and can be regarded as a refinement [23] The approach is in line with the basic trends presently being developed for designing automatic control systems for sophisticated power-production complexes [24].

REFERENCES

1. A. A. Shalyto, "Algorithmic graph schemes and transition graphs: their use in software realization of logical control algorithms. Part I," *Avtomat. Telemekh.*, No. 6, 148–158 (1996).
2. S. I. Baranov, *Synthesis of Micro-Programmed Automata (Graph Schemes and Automata)* [in Russian], Energiya, Leningrad (1979).
3. A. A. Shalyto, *Realization of Algorithms for Marine Logical Control Systems by Microprocessor Technology* [in Russian], Advanced School for Managers and Marine Engineers, Leningrad (1988).
4. A. A. Shalyto, *Software Realization of Logical Control Algorithms for Marine Systems* [in Russian], Advanced School for Managers and Marine Engineers, Leningrad (1989).
5. A. A. Shalyto, "Software realization of controlled automata," *Sudostroitel'naya Promyshlennost', Ser. Avtomat. Telemekh.*, No. 13, 41–42 (1991).
6. A. A. Shalyto, "Technology for implementing logical control algorithms as a tool for enhancing longevity," in: *Abst. Conf. Longevity of Ships and Vessels*, Sudostroenie, Saint Peterburg (1992), pp. 87–89.
7. V. V. Rudnev, "Interconnected graph systems and algorithmic programming of discrete controllers," *Avtomat. Telemekh.*, No. 7, 110–121 (1979).
8. V. I. Rubinov and A. A. Shalyto, "Construction of graph schemes of binary programs for a system of Boolean functions defined by truth tables," *Avtomat. Vychisl. Tekh.*, No. 1, 87–92 (1988).
9. G. Mishel, *Programmable Controllers. Their Architecture and Applications* [Russian translation], Mashinostroenie, Moscow (1992).
10. D. Jamp, *AUTOCAD Programming* [Russian translation], Radio i Svyaz', Moscow (1992).
11. L. Zadhe and C. Desoer, *Theory of Linear Systems. The State Space Method* [Russian translation], Nauka, Moscow (1970).
12. *Autolog 32. User's Manual*, FF–Elektroniikka Fredriksson Ky (1990).
13. E. Iodan, *Structural Design and Construction of Programs* [Russian translation], Mir, Moscow (1979).
14. R. Linger, K. Mills, and S. Whitt, *Structural Programming: Theory and Practice* [Russian translation], Mir, Moscow (1982).
15. F. Schmalhofer and J. Thoben, "A model-based construction of a CASE-oriented expert system," *Eur. J. Artif. Intel.*, **5**, No. 1, 38–45 (1992).
16. "CASE-Technology in Russia: Present or Future?" *Computer World, Moscow*, No. 40–41, 8–9 (1992).
17. N. A. Krinitskii, *Algorithms Around Us* [in Russian], Nauka, Moscow (1984).
18. *Functional Description. Warm-up & Prelubrication Logic. Generator Control Unit*, Severnaya Hull No. 431, NORCONTROL, Norway (1993).
19. *SELMA 2. Monitoring System. Programming and Operation Guide*, RU 5351265-8C, ABB ASEA BROWN BOVERI, STROMBERG, Finland.
20. A. M. Gel'fand, V. N. Shumilov, I. D. Ablin, et al, "TEXNOKONT—A multifunctional package of software and hardware tools for constructing distributed control systems," *Prib. Sist. Upravlen.*, No. 1, 2–9 (1994).
21. O. P. Kuznetsov, A. Ya. Makarevskii, A. V. Markovskii, et al., "YARUS—A language for describing the operation of complex automata," *Avtomat. Telemekh.*, No. 6, 80–89 (1972); No. 7, 72–82 (1972).
22. O. P. Kuznetsov, L. B. Shipilina, A. V. Markovskii, et al., "Design of logical programming languages and their computer-aided realization (in YARUS-2)," *Avtomat. Telemekh.*, No. 6, 128–138 (1985).
23. A. A. Shalyto, "Cognitive properties of hierarchical representations of complex logical structures," in: *Proc. 1995 Int. Symp. Intel. Contr, Workshop*, Monterey, California (1995).
24. I. V. Prangishvili and A. A. Ambartsumyan, *Principles of Construction of Automatic Control Systems for Transition Tables of Sophisticated Power-Production Systems* [in Russian], Nauka, Moscow (1992).