Automation and Remote Control, Vol. 62, No. 1, 2001, pp. 1–29. Translated from Avtomatika i Telemekhanika, No. 1, 2001, pp. 3–39.

REVIEWS

Logic Control and "Reactive" Systems: Algorithmization and Programming

A. A. Shalyto

Federal State Science Center Avrora, St. Petersburg, Russia
State Institute of Fine Mechanics and Optics, St. Petersburg, Russia

Received June 13, 1999

Abstract—Algorithmization and programming principles for logic control and "reactive" systems are formulated, regarding algorithms and programs as finite automata. The application of finite automata to programming for other problems is also reviewed.

1. INTRODUCTION

Finite automata, which in the past were mainly used in hardware, are presently finding extensive application in programming. In this paper, we outline the basic algorithmization and programming principles for logic control and "reactive" systems, regarding programs as finite automata. Besides the traditional fields of application, such as compiler design, finite automata are presently used in programming programmable logic (PL) controllers, describing the behavior of certain objects in object-oriented programming, as well as in programming protocols, games, and PL circuitry.

We would like to dispel the misbelief that "automata are becoming obsolete" and show they are just beginning to find extensive application in programming.

2. ALGORITHMIZATION AND PROGRAMMING FOR LOGIC CONTROL SYSTEMS

The International Standard IEC 1131-3 [1] establishes programming languages for PL controllers and PC-controllers—industrial (control) computers (usually IBM PC compatibles) with SoftPLC and SoftLogic application software. There also exist programming languages for microcontrollers and industrial (control) computers [2]. According to [8], thus far no algorithmization language for logic control problems (based on true and false logic) has been developed that might be helpful in

—understanding what has been done, what is being done, and what must done in a programming project in different fields of knowledge,

—formally and isomorphically converting an algorithm into programs in different programming languages with a minimal number of internal (control) variables, because these variables hinder clear understanding of programs,

—easily and correctly modifying algorithms and programs,and

—correctly certifying programs.

Since such a language is not available, thus far there is no algorithmization and programming technology that might enhance the quality of software for logic control systems.

Review [8] surveys the well-known algorithmization and programming technologies for logic control and reactive systems, i.e., the technologies underlying the new switch-technology meeting the above requirements, which can also be called the state-technology or, more exactly, the automaton-technology. We describe its basic principles.

1. "Internal state" (simply, state in what follows) is the basic concept in this technology.

0005-1179/01/6201-0001$25.00

State is regarded as a kind of abstraction introduced in algorithmization via one-one correspondence of a state with one of the physical states of the controlled object, because "operation of a production system is usually manifested as variations in its states" [9]. Each state in the algorithm maintains the object in the respective state and a transition to a new state in the algorithm corresponds to a transition of the object to the respective state, thereby implementing the logic control.

For example, a valve may exist in one of the four states (closed, opening, open, and closing), each of which is maintained by the corresponding state in the control algorithm. For a memory-valve, the control algorithm may also have a common state for maintaining its closed and open states [8]. The control algorithm may also contain, if necessary, other states related to valve defects and operator faults.

The relationship of the states with internal (control) variables is manifested at the state coding stage, which is absent in traditional programming. The number of control variables depends on the coding scheme.

The approach used in automaton theory is essentially different from the approach usually used in programming, in which internal (usually, binary) variables are introduced, if necessary, in the course of programming and then each set of code values is declared a state of the program [10]. Since the "state" concept is usually not used in application programming, the number of states in a program containing, for example, n binary interval variables in most cases is obscure. However, the number of states may vary from n to 2^n. It is also not clear where do these variables come from, how many variables are there, and the purpose for which they are applied. In cyclic implementation (due to output-input feedback), a program may also operate sequentially even in the absence of control variables [8].

2. Along with the state concept, the concept of "input" naturally requires the concept of an "outputless automaton" (outputless automaton = states + inputs). If the concept of an "output" is defined, by which we mean "action" and "activity," this formula reads automaton = states + inputs + outputs. The corresponding programming field can be called the "automaton programming," and program designing can be called the "automaton program designing." As in [11, 12], action is single-time, instantaneous, and continuous, whereas an activity may last long and interrupted by some input. An input in general may change the state and initiate output with or without change of state.

3. The main model underlying the automaton technology is a "finite deterministic automaton" (in the sequel, simply automaton), which, according to the classification of [13], is an automaton with internal states, but not an automaton with behavior functions [8].

4. Automata without output converters (Moore machines), Mealy machines, and combined machines (C-machines or Moore–Mealy machines) [14, 15] are used as structural models in [8].

The main structural model is defined by Moore machines, in which state codes and output values are distinct and the values of output variables in each state do not depend on inputs. This simplifies the introduction of changes in their description. In [8], "controllability" is defined to be the properties of algorithms and programs that aid in their correction.

Initially, the number of states of a Moore machine can be chosen equal to the number of states of the controlled object (including its faulty states, if necessary). Subsequently, additional states related, for example, to operator's faults may be included [8]. Then the number of automaton states can be minimized by combining equivalent states or using some other structural model and this, unless absolutely unavoidable, is rather undesirable.

In a Moore machine, the values of output variables are retained in the memory realizing these states as long as the machine exists in the corresponding state. In a Mealy machine, these values are also formed in the corresponding transition and, consequently, an additional memory different

from the state-realizing memory is required for long storage of these values. In this respect, Moore and Mealy machines are not equivalent.

5. Automata of the first and second kind are also used [8, 16]. Preference is given to the latter, in which a new state and output values are formed without delay in a "cycle" (in one program cycle).

6. The automaton states are encrypted by different coding schemes [8], for example, forced, free, binary, binary logarithmic, and multivalued codes. Multivalued state coding is preferable, because it can be used for the states of automata (due to the presence of only one active state in them) and the traditional viewpoint that automata are particular Petri networks can be discarded, because a single variable cannot be used for state coding for the reason that several states are concurrently active in a Petri network. While only one multivalued variable is sufficient to distinguish the states of an automaton regardless of the number of states, the number of binary variables required to distinguish the positions equals number of states in a Petri network in which not more than one label can exist in each position (analog of the Standard NFC-03-190 Grafset). As shown in [8], a Grafset diagram with parallel "segments" can be replaced by a system of interconnected transition graphs. In the first case, while the number of binary variables is equal to the number of positions, the number of multivalued variables for the second case is equal to the number of transition graphs, irrespective of the number of vertices in them.

7. The nonprocedural visual formalism of automaton theory, such as transition graphs (state charts or state transition charts) is used in programming the algorithmic model of finite-memory automata.

In the names of these terms, as in automaton technology, preference is given to the concept of "state," rather than to the concept of "event" commonly used in modern programming. In our approach, the concept of an event is only secondary and, along with the input variable, is regarded as a modification of the input that may change the state.

The procedural visual formalism (graph-schemes of algorithms and programs) elaborated in theoretical programming does not use the "state" concept in explicit form and this complicates the understandability of this concept [17]. This concept is also not used in the language of regular expressions of event algebra [18].

It is easy to recognize transition graphs as they are planner and have no height (like algorithm schemes, SDL- [8], and Grafset diagrams). They are far more compact compared to equivalent schemes consisting of functional blocks (traditional logic elements) and easily comprehensible, because interaction between transition graphs is implemented with data and the interaction between schemes is implemented by control.

8. An advantage of transition graphs, which must be "maximally" planar, is that every arc shows not all inputs (as minterms), but only those that ensure transitions along this arc. The inputs on every arc can be combined into Boolean formulas of arbitrary depth. Therefore, the description of the algorithm is highly compact.

The use of Boolean formulas in a model, as in hardware realization (structural synthesis of series circuits), widens the classical abstract finite automaton model, in which arcs are labeled only with the input alphabet characters, and aids in "parallel" processing of inputs. Compared to input-sequential program realization, such a description of algorithms in general decreases the number of states in automata generating these algorithms [8].

We can assert that every state in a transition graph "identifies" from the set of all inputs only the subset that ensures transitions from this state, i.e., decomposes the input set.

Therefore, transition graphs can be applied in solving logic control problems containing a large number of inputs, thereby simplifying comprehensive testing along all routes.

9. Every transition graph must be semantically and syntactically correct. The first property determines whether the correct model has been designed, whereas the second property determines whether the model has been designed properly [19]. In verifying the syntactical correctness, a transition graph is tested for attainability, completeness, consistency, realizability, and absence of generating circuits. In testing attainability, the presence of at least one path from every vertex to any other vertex is verified. Completeness [8] is verified for every vertex and is useful in discarding loop labels, in particular, for Moore automata. If every vertex is consistent [8], priorities can be used instead of orthogonalization to reduce the program realization complexity. Realizability is ensured by different identically labeled vertices. Generating circuits [8] are eliminated via orthogonalization of labels of arcs forming the corresponding circuits.

10. Further refinement of the abstract finite-automaton model and output "parallelism," as in hardware implementation, are attained by indicating on vertices and/or arcs the values of output variables (activities) formed at these graph components. Display of the values of all output variables at each component (depending on the automaton structural model) aids in understanding the algorithm described by a transition graph due to the increased number of vertices. The transition graph thus constructed defines a structural finite automaton without any omissions of the values of output variables. For the transition graph of a Moore automaton, the values of output variables that do not change in successive transitions and shown at vertices at which transitions occur must be commented out to reduce the program size.

The finite-automaton model can be improved further by (at the cost of understandability) by indicating both the values of output variables and Boolean formulas (automaton formulas for embedded automata) at graph components [8].

11. Input and output "parallelism" aids in realizing parallel processes even by a single automaton, which, by definition, is a sequential state machine (only one vertex of the corresponding transition graph is active at an instant).

12. Unlike a memoryless automaton, the behavior of a finite-memory automaton depends on its prehistory: every transition to a state depends on the preceding state, whereas outputs of a Moore machine depend only on the state in which the machine exists.

In the automaton technology, these properties of finite-memory automata must be preserved and the transition graphs must not contain any flags and omissions to eliminate the dependence on state and output prehistory, respectively [8, 17]. The transition graph must have as many vertices as states of the automaton. Prehistory independence (future depends on the present and does not depend on the past) aids in understanding the automaton behavior [20] and introducing changes in the transition graph and its realization program.

A similar situation is also observed in Markov processes, which can be investigated by a simple tool, which may become unwieldy if the process is prehistory-dependent.

13. In formal and isomorphic transformation of a transition graph without flags and omissions to a program, the graph is the specification test for the program. Here there is a possibility for certification, beginning from the convolute of every transition graph with its isomorphic program fragment. Testing and verification based on "finite-state machines" or "state machines" are investigated in [21–28]. An approach based on an extended finite-automaton model and used in "VisualState" software to verification of built-in programs is described in Computer Journal, no. 5, 2000. This approach is used to verify the attainability of implicitly defined states of a system.

14. One internal variable of significance digits equal to the number of states is sufficient for coding the states of a finite-state automata with multivalued codes.

Since (in principle) any logic control algorithm can be realized by one transition graph, the graph can be realized by a program containing only one internal (control) variable, regardless of the number of vertices in the graph.

In every transition, the previous value of the multivalued variable is automatically discarded and this variable is unrivaled.

Such a coding scheme can be applied only if the number of states of the automaton are known at the beginning of the algorithmization process.

15. Almost all programming languages, even the exotic functional block language, can handle multivalued variables [8, 29].

A transition graph with multivalued coding for vertices corresponding to the automaton model is formally and isomorphically realized by one or two switch constructs of C [8] or their analogs in other programming languages. This explains the name of the technology. Moreover, the word "switch" is associated with the switching circuit theory—the base of logic control theory.

Transition graphs can be realized via two approaches, which can be called the "state–event" and "event–state" approaches. In the first approach, a transition graph is realized by one or two "state" switch constructs, whereas in the second approach, each "event" is associated with a function described by a "state" switch defining the transitions initiated by this "event" [30]. If events are represented by the values of a multivalued variable (function forming these values), then the state switch embedding event switches is the primary construct in the first approach, whereas the event switch embedding state switches is the primary construct in the second approach [8]. In the first approach, programs are designed via logic control principles (state-state transition under the action of an event) to make programs user-friendly. In advanced disciplines (for example, physics), the concept of "state space" is fundamental [31], whereas the concept of "event flow" is secondary. For example, (liquid, solid, gas) states of water are decisive, whereas state-state transition conditions and water interactions are secondary.

16. The value of a multivalued variable is characterized by the "position" of the program realizing the transition graph in the state space. Therefore, the concept of "observability" (for one internal variable) can be introduced in programming, regarding the program as a "white box" with one internal parameter.

A formal and isomorphic program designed from the transition graph of a Moore machine without flags and omissions can be verified by testing the realization of all transitions by observing the values of only one internal variable and testing the values all output variables in each state.

Transition graphs of other types of automata can always be transformed into transition graphs of an equivalent Moore machine. For a Mealy machine, each vertex of its transition graph without flags and omissions is associated with a vertex of the transition graph of an equivalent Moore machine. The number of vertices is equal to the number of incoming arcs at the vertex and labeled by different values of output variables [8].

The functionalities of a transition graph can be studied with an attainable label graph, i.e., a reachable state graph, in which each vertex may be uniquely associated with an activity. Therefore, the transition graph of a Moore machine (an automaton without any output converter) without flags and omissions can be used as the reachable state graph, whereas transition graphs and attainable state graphs of other types of automata differ. For example, the transition graph of an equivalent Moore machine can be used as the transition graph without flags and omissions for a Mealy machine.

17. From the foregoing, it follows that our approach is helpful in solving the problem, called decoding [32] or recognition [33] or identification [34] of automata, because an automaton is recognized (decoded, identified) as soon as its transition graph is constructed.

An automaton is not recognizable if it is an "absolutely black box" for which no information on its internal states is available [32]. For finite-memory automata, "input–output" tests used in designing logic control systems are helpless in recognition and guaranteeing a predefined behavior of a system [35].

A finite deterministic automaton can be recognized by its "input–output" pattern if the maximal number of states is known in minimal (number of states) form and its transition graph is strongly connected [33].

An automaton with known estimate of the number of states is called a "relatively black box" [32].

18. Thus, it is not possible to recognize automata via testing algorithms and programs by their input–output patterns if the "state" concept is not defined.

There is no problem of recognition in the automaton technology, because the state concept is defined and transition graphs are strongly connected.

In the switch-technology, "black boxes" are discarded, but "white boxes" are used. An automaton defined in any form different from a transition graph without flags and omissions is called the "relatively black box" and an automaton defined by a transition graph without flags and omissions is called the "absolutely white box."

A "relatively black box" can be recognized via mathematical transformations [8].

19. In the switch-technology, a system of interconnected transition graphs is used as the algorithm model [36] to support the possibility of composition and decomposition of algorithms and ensure practical application in designing complex logic control systems.

Moreover, automata (transition graphs) can generate centralized, decentralized, and hierarchical structures [8].

20. If a system contains N transition graphs with arbitrary number of states, then only N internal multivalued variables of the state coding scheme can be used in programs even with regard for the interaction of graphs.

For this purpose, the program realization must be constrained: only one transition in every transition graph must be implemented in a program cycle. Thus, the previous state of the automaton is preserved or only one transition to this state is implemented even if several transitions may occur successively in a program cycle in this automaton.

This constraint ensures the attainability of every value of the internal variable for other $N - 1$ graphs of the system, Thus, the number of internal variables is not increased for implementing the interconnection of transition graphs and the interaction of transition graphs is made effective through multivalued internal variables. The predicates of the verified values of these variables are used as a modified input.

Both in automaton and object-oriented programming, automata exchange messages. As a result of the possibility of interaction between automata constituting a program via exchange of the numbers of their internal states, the automaton programming differs from the object-oriented programming, in which objects are regarded as "black boxes" with encapsulated internal contents [37].

21. Transition graphs of a system may interact according to the "inquiry–answer–deletion" or "inquiry–answer" principle by exchanging the numbers of states.

A particular case of such an interaction is the interconnection between the main and other transition graphs. If necessary, here there is a possibility for starting parallel processes realized by these graphs from the main graph and for returning the control to the main graph upon completion of work. This widens the field of application of finite-state automata, which, by definition, are sequential state machines. Algorithms implemented as a system of interconnected transition graphs are superior to the Grafset language developed by Telemechanic (France) division of Scheider Electric for describing series-parallel processes [9].

22. Besides the exchange of the numbers of states by automata generated by sequential switches in a program, transition graphs may also interact according to the embedding principle. This

can be implemented by embedded switches of arbitrary depth or functions constructed from these switches that realize transition graphs of a specified embedding depth.

23. These graph-graph interaction principles support hierarchical algorithm design.

In automaton technology, "up-down" and "down–up" algorithm designs are possible. In the "up-down" strategy, it is easy to design correct algorithms. For example, if the algorithm is not state-parallel in design, every vertex in the initial transition graph can be replaced by a fragment containing a few vertices, other transition graphs can be nested in this fragment, and other transition graphs can be accessed from this fragment. Every vertex in the structure thus obtained, in turn, can be refined. The program realizing this algorithm must be isomorphic to the algorithmic structure.

24. For an algorithm designed as a system of transition graphs, this system, if possible, must be decomposed into disconnected subsystems formed by interconnected transition graphs and a graph of attainable labels (states) describing the functionalities is constructed for each subsystem, if dimension permits.

Thus, in the automation technology, if the subsystems of transition graphs are of suitable dimension, a system of transition graphs and their programs can be tested by their functionalities, as in verifying protocols [38].

If strongly connected systems of high-dimensional transition graphs are used, it is possible in automaton technology to design "verifiable" programs via automatic protocolling of program operation in terms of automata [39].

25. A control automaton is defined as a set of an automaton and functional delay elements. These elements are regarded as a controlled object (servos and signalizers): along with the values of 'object" output variables, the automaton also generates the values of "time" output variables. Along with the values of "object" input variables, the automaton receives the values of "time" input variables. Both "time" and "object" variables are shown on transition graphs. Different approaches to program realization of functional delay elements by functions are examined in [8].

Along with such functions in the components of transition graphs, other types of functions realizing controllers may also be used [8].

26. Transition graphs can also be used in designing models for logic control objects and describing and modeling both open and closed "control algorithm–model of controlled object" complexes from a unified standpoint.

27. In the automaton technology, the control program specifications must consist of a closed "data source–system of interconnected automata–functional delay elements–servos–data representation media" connection scheme describing the interfaces of automata and a system of interconnected transition graphs describing the behavior of automata in the scheme. The connection circuit (some analog of the data flow diagram used in structural system analysis [40]) and transition graphs must be displayed compactly (whenever possible) to aid in understanding the specifications (the Gestalt description requirements [41]). In particular, transition graphs must contain only symbols of variables, but not acronyms, and the meaning of every variable must be clear in viewing a connection scheme, which may also contain comments.

For large-dimensional problems, this circuit can be constructed (for the whole "environment") separately for each automaton.

28. The switch-technology permits only one language for algorithm (transition graph) specifications under different programming languages. This point escaped notice in [1].

Methods of realization of formal and isomorphic transition graphs by programs in different languages used in control devices, including PL controllers, are described in [8]. Analogs of switch constructs helpful in designing compact and comprehensible program texts are best suited for PL

controllers. This construction is modeled by digital multiplexers ensuring the possibility of designing PL controllers for functional circuits formally and isomorphically realizing different transition graphs [8].

Transition graphs designed as above and used as program design specifications (defining functional capabilities) and as specification tests can also be written in Grafset. Here there is a possibility of visually observing transitions in graphs. But this language has certain disadvantages, for example, concurrent display of the activities of several vertices in one diagram results in binary coding, in which the number of binary internal variables (used in coding vertices and in PL controllers in limited number) is equal to the number of vertices in the diagram. The Grafset language can only realize transition graphs of Moore machines, increasing the number of vertices in the transition graphs corresponding to other types of automata. Every transition graph is displayed on several screens and back-transition graphs are invisible, because lines joining vertices are not displayed or cannot be displayed and are, therefore, replaced by "links."

While a PL controller in realizing a set of transition graphs utilizes all binary internal variables allotted for programs in Grafset, other types of internal variables of the controllers can be used if the programming language is changed for realizing another set of transition graphs. Thus, even in programming one PL controller, a need may arise for different programming languages in applying the same specification language.

29. The automaton approach is helpful to the customer, designer, technologist, user, and controller in understanding what has been done, what is being done, and what must be done in the program project. It is useful in distributing the work and, most importantly, the responsibility between different specialists and organizations. Such a division of labor is pivotal particularly in projects with foreign participation due to language barriers and misunderstanding.

The approach is helpful in handling project details even at the early design stage and in demonstrating them to the customer in a convenient form.

30. This technology permits communication between designers in terms of technological processes (for example, emergency start of the diesel-generator is not functioning) in a formalized and understandable language (a sort of technical Esperanto in which they can communicate, for example, as follows: "change the value from zero to one at the fourth position at the fifth vertex in the third transition graph") to avoid misunderstanding due to confusion even within one language and when participants of different countries are involved in a project and no specialist knowledgeable in the technological process is required to introduce changes in the program [42].

31. In this approach, the programmer need not know the technological process and the designer need not know the details of programming. The application programmer need not do the jobs of the customer, technologist, and designer, but restrict himself to the realization of formalized specifications. Thus, the fields of knowledge he must possess can be narrowed and ultimately his job can be automated. The "traces" of work of one designer can be retained so that program support and modification can be done by some other specialists. It is also effective in controlling the design and text of programs, not just the results as in most cases. Thus, approval of programs can be replaced by the approval of equipment, which consists not in testing only [8].

32. In implementing transition graphs, the values of "time" output variables are replaced by functions realizing functional delay elements and the values of "object" output variables can be replaced by functions of other types. Therefore, this approach can also be applied in implementing the control part of logic algorithms, as demonstrated by the program compiled [8] for synchronizing the generator with the main distribution board bus.

33. The automaton technology was successfully applied by Avrora Inc. (St. Petersburg) in designing control systems for ship equipment and other projects based on diverse computing apparatus employing different programming languages.

In using one specification language, PL/M language [42], functional block language [43], assembler language of single crystal KP 1816 BE 51 PC [44], and ALPro instruction language were used for designing complex control systems of Ship Project 17310 [45, 46].

To simplify programming for Project 17310, B. P. Kuznetsov (Avrora Inc.) jointly with me designed a "C core–ALPro" [47] translator for formally and isomorphically compiling programs in C from transition graphs.

34. Design, testing, and maintenance of several logic control systems have corroborated the effectiveness of switch-technology, at least, for the systems under review. According to Norcontrol (Norway) [42], this approach produced a high quality logic control system for the Marine diesel generator DGR-2A 500∗500 of Project 15640. Its application to other control problems is described in [8].

35. Thus, this technology is very effective for algorithmization and programming [42].

36. This technology can be characterized by seven parameters: state, automaton, prehistory-independence, multivalued state coding, a system of interconnected transition graphs, formal and isomorphic programming, and switch constructs. The formal and isomorphic programs constructed from transition graphs by this technology are demonstrative, comprehensible, structurable, addressable, embeddable, hierarchical, controllable, and observable.

The relatively small number of internal variables resulting from the application of multivalued state coding considerably simplifies and speeds up shipping of states in computations, for example, in designing robust systems in which trigger contents are shipped infinitely many times if the functional block language is applied.

37. In other problems, the states of the control automaton described by a transition graph must be distinguished from the memory states. While the number of automaton states is usually not greater a few tens, the number of memory states is far greater than this amount [18]. Therefore, they are not identified explicitly. If the states are not partitioned in this manner, as in [37], states are not used and the program behavior is determined as a set of actions in response to events without reference to the automaton states with which these events are associated. Furthermore, the control variables must be distinguished from the internal variables of other attributes.

38. In this technology, control automata can be designed for individual modes (gang valve opener and closer), combined modes (close–open automaton for a valve gang), and individual objects (valves) for implementing individual or combined modes.

Automata may interact between themselves by exchanging the number of states, nesting and addressing, and via auxiliary automata. If a system contains many automata, it is difficult to demonstrate the correctness of their proper joint operation. Therefore, the automaton technology can be applied both in object-oriented and procedural designs [8].

39. If the initial algorithm is realized by a single-input single-output graph scheme [48], it can be applied jointly with the methods of [49] to construct the transition graph of the automaton [8, 50] realized by a switch construct and DO WHILE operator with a condition defined by the number of the terminal vertex. This method is more effective than the Ashcroft and Mann method [8]. By way of example, a transition graph realizing a number-sorting algorithm described by a cyclic graph scheme is given in [51].

40. In nonprocedural specification of automata as transition and output tables, rows and columns (or vice versa) define states and events. Therefore, processing sequence (state–event or event–state) for the interpreter in nonprocedural realization of automata is immaterial [8]. In procedural specification of automata as algorithmic or program schemes [52], realization is procedural, requires less memory, and depends on the state and event processing sequence.

The schemes with an event (values of input variables) decoder are called event schemes and the schemes a state decoder are called automaton schemes. Their design approaches are called the event and automaton approaches, respectively.

In the automaton approach, there is no need to use these schemes, because programming can be done via transition graphs (with multivalued vertex coding), which are isomorphic to the state-decoder schemes (automaton scheme of algorithms), and C switch constructs [8, 17].

Algorithm schemes constructed via event and automaton approaches are designed in [8] for realizing R- and counter triggers, respectively. They are more "understandable."

Programs for realizing certain control functions for GUI toolbar elements and constructed via event and automaton approaches are described in [53, 54]. In the event approach, as in object-oriented programming in which emphasis is laid on autonomous interacting agents for attaining the desired result [37], a separate handler is created for each event. As in object-oriented programming, interaction between methods is ensured by heuristic flag (control) variables (one in our case) without any regard for the understandability and correctness of the program sequential logic. In the automaton approach, event handlers call a function for realizing a formal and isomorphic transition graph by transferring the number of event as a parameter to the function. Therefore, the logic, which is originally distributed among handlers, is now crowded in one function to improve the understandability of the behavior of the program [54, 55].

41. This technology is complete and transparent, because it embraces all stages of algorithmization and programming for which methods guaranteeing high-quality for the design of a project as a whole are available.

42. A detailed description of this approach is given in [8, 17, 56–59]. Similar approaches are described in [36, 60–62].

This approach augments the International Standard IEC 1131-3 (IEC 61131-3) [1], which does not establish (unlike [8]) methods of designing programs in PLC languages. This topic is also not covered in the documents of major automation companies [63], which only give examples [64, 65–74].

This approach was developed in 1991 [43, 75] and applied at Avrora Incorporation for designing the following equipment.

—Control systems for the diesel generator DGR-2A $500 * 500$ fitted in three vessels of Project 15640 based on the equipment of "Selma-2" and ABB Stromberg (Finland). The program was complied in the functional block language [43].

—Control system for this diesel-generator for the vessels of Project 15967 based on "Selma-2" equipment of Stromberg (Finland). The program was compiled in the functional block language.

—Control system for this generator for the vessels of Project 15760 on the basis of the equipment of Norcontrol (Norway). The program was compiled by Norcontrol in PL/M [42].

—Control system for the technical equipment of five vessels of Project 17310 based on Autolog controllers of FF-Automation OY, Inc. (Finland) [45]. Programs for the general equipment were compiled in ALPro [46] and for the control system of auxiliary mechanisms with the "C core–ALPro" translator [47].

—Avrolog control complex for the ship technical equipment were developed on the basis of Autolog controllers of FF-Automation OY. Program was compiled with the "C core–ALPro" translator [47].

—Automatic control system for the technological processes of central Primary Petroleum Processing Station Avrolog-NP1 (Severo-Orekhov oil wells). Program was complied in ALPro.

—Automatic control system for the technological processes of the compression pump station Avrolog-NP2 (Severo-Pokursk oil wells). Program was written in ALPro.

—Control system for the turbocompressor Larina at Polimer plant (Novopolotsk). Program was compiled in the assembler language of the single-crystal microcomputer KP 1816 BE51 [44].

3. APPLICATION OF FINITE AUTOMATA FOR PROGRAMMING PROGRAMMABLE LOGIC CONTROLLERS

Modicon (USA), a sister division of Schneider Automation [4], in 1993 and Siemens (Germany) in 1996 elaborated transition graphs as a programming language for their PL controllers. According to Siemens [2], description in this language is helpful not only for PLC programmers, but also understandable to engineering personnel. They developed this language for design purposes, though it is not specified in IEC 1131-3 [1].

These approaches are better than the automaton technology in that the translators are written in the transition graph language and an executable specification language is developed [76]. Their disadvantages are the translators are not interchangeable, the translators restrict the application of models and approaches other than those specified in the design, documentation does not describe the transition graph design specifics that aid in understanding and quality of programs, there is no mention to the need for designing connection schemes, graphs are difficult to view, because graphs, transition conditions, and output variables are displayed on different screens (non-Gestalt description), and transition graphs are used as a programming language without any mention that transition graphs can also be used as an algorithmization language along with other programming languages.

At present, the transition-graph approach, called the State Logic, is also used by General Electric Fanuc Automation for programming its old controller models [77]. ECLPS (English Control Language Program Software) has been developed for describing transition graphs. GE Fanuc Automation believes that this approach can be used without programming experience and is an alternative to ladder schemes.

The disadvantages of this approach are the following: transition graphs are represented as labeled with English words, the special programming language uses English language and is not based on mathematical notation to avoid omission, it requires a special processor (State Logic Processor), and transition graphs are not recommended as a specification language for programming in ladder-scheme language of older PL controller models.

State diagrams are used as a programming language for industrial controllers produced by Matsushita Automation Controls [78] and the "state list" language is used as the programming language for the controllers manufactured by Festo Cybernetic [79].

4. APPLICATION OF FINITE AUTOMATA IN PROGRAMMING

Until recently, transition graphs were used only in theoretical programming problems [80] and in practice they were mainly used only in designing compilers [81, 82], though transition graphs have been used since long in hardware realization [16]. Transition graphs are used in programming, because "any program can be regarded as if it has been realized by hardware equipment" [83].

Until the last decade, the main tool for designing programs for functional (applied) problems was the approach based on transition graphs and visual formalism (algorithmic schemes).

Limitations (in the opinion of Booch [84]) of this approach in designing complex programs stimulated the development and wide use of the object-oriented approach. But this new approach in incipient stages was supported only by programming languages [85] and there was no program design method within the framework of this paradigm.

The method developed by Booch in 1991 is based on several visual formalisms (diagrams) for displaying different properties of selected classes and objects [84]. He also applied transition dia-

grams for describing the behavior of objects and illustrated the application of these diagrams with examples [84]. Unlike in our approach, the diagrams in the Booch method are only word-labeled "charts" [12], but not mathematical models. Moreover, Booch failed to notice (unlike in [8, 17]) that these charts under a certain approach to designing algorithmic schemes might be isomorphic to transition graphs and, consequently, the comparison in this case is incorrect. This fact also escaped mention in [50, 86].

A situation similar to that of Booch is observed in [87], in which the "world" is modeled in state. This "world" mainly consists of microwave ovens, valves, and pumps that are logic controlled objects.

Booch pays great attention to state and transition diagrams without modifying the definition of a state in terms of internal variables in [11], which is based on the works of Harel [88, 89], who, in the opinion of Booch, elaborated a "simple but very expressive approach far more effective than the traditional finite-state automata." But this comparison is rather inappropriate, because Harel's approach, like our approach, is based on finite-state automata.

Harel developed his approach for control systems, called reactive systems [90], which are slightly wider than logic control systems and their behavior is best characterized by their response to events taking place beyond context [91]. Such systems respond to a flow of events by changing the states and actions in state-state transitions or actions and activities in states [92–100]. He also developed a visual formalism (modified state and transition diagram), called the "state-chart" [88, 92], which is mathematically equivalent to the Moore–Mealy automaton diagram [90], but describes these automata more compactly in certain cases. State charts can also be called the Harel diagrams and their main features are the following.

Along with states, hyperstates (superstates [12]) consisting of several states that respond identically to an event [101] can also be used. Instead of displaying transitions to a state contained in a hyperstate, only one transition from the hyperstate to a state (generalized transition [101]) is displayed.

Hyperstates may theoretically have any embedding depth. Transitions from a hyperstate involve all embedding levels in a hyperstate.

A hyperstate may consist of OR-states (sequential states) or AND-states (parallel states) [91]. In the first case, an automaton on passing to a hyperstate can exist only in one state (or another), whereas in the second case, a hyperstate must contain parallel states in orthogonal domains [91].

Diagram vertices corresponding to states are hyperstates and are labeled with activities (do), single actions implemented on entering a vertex (entry), and single actions on exiting a vertex (exit).

The last two cases can be called the generalized actions, because single actions implemented upon entry replace identical actions that are labeled in all incoming arcs of the corresponding transitions in a Mealy or combined automaton, and single actions implemented upon exit replace identical actions that are labeled in all outgoing arcs of corresponding transitions in a Mealy or combined automaton.

An arc may be labeled with a transition-initiating event and the transition-"preserving" logical condition. An arc may also be labeled with single actions implemented in a transition or events formed in a transition.

Diagrams may also display pseudostates, for example conditional (C), terminal (T), and historical (H) states, which are not real states.

If a hyperstate contains a historical pseudostate, then in transition to this hyperstate, control is transferred to the state in which the system last existed in this hyperstate [101]. The use of generalized transitions and historical pseudostates is helpful in effectively specifying interruption

due to transition of the system from a state belonging to a hyperstate to another state in which interruption is processed and operation of the system is continued (after processing of interruption) from the state in which interruption occurred [101].

Certain ideas of Harel, for example, generalized transitions, can also be applied in the switch-technology.

Though the Harel approach is somewhat similar to our approach, they are quite distinct. First, both states and hyperstates can be used in the Harel approach, whereas only states are used in the switch-technology. Second, in the presence of a hyperstate, at least, with OR-states, it is believed that the Harel diagram describes the behavior of only one automaton, whereas a system of interconnected transition graphs describing the operation of a system of interconnected automata can be used in the switch-technology.

Moreover, it is not clear how the Harel notation can be used to display a diagram for a large number of hyperstates with a large embedding depth. For this reason, the Harel diagram (state diagram) in object-oriented modeling is believed to describe the lifecyle of an individual object and the behavior of a community of jointly operating objects is modeled by some other diagram (interaction diagram) [91] and this complicates the formal approach to realization.

The described technology uses the concept of an "automaton" and dynamic charts of one type (a system of interconnected transition graphs) even in the presence of a large number of automata with a large embedding depth, i.e., charts can be formally and isomorphically realized in any programming language.

Moreover, the formal relationship between static and dynamic charts in the object-oriented approach is obscure [90, 91]. But the concept is distinctly expressed in terms of automata if the object diagram is replaced by an automaton interaction scheme and a connection scheme containing a system of interconnected automata and a system of interconnected transition graphs is used.

For this reason and its complexity and specifics of notation, Harel diagrams are thus far not used in programming for programmable controllers.

The switch-technology can be used as a software designing tool, at least, for SoftLogic [72]. This approach does not say anything about whether the software works; it only explains why a program works [20]. It, in particular, answers where do internal (control) variables come from?, how many variables must be used?, and what for is a variable used?.

Automaton approach is used in the specifications of computer network protocols [38, 102–105] and in the specification and description language (SDL) [106–108] developed by the International Commission for Telephony and Telegraph for designing software for telecommunication systems. But the SDL-diagrams, being a modified algorithm scheme in which states can be explicitly introduced, are unwieldy and applicable only to Mealy automata. Moreover, SDL-diagrams disregard omissions and flags. Other disadvantages of SDL-diagrams (their height narrows the vision range of specifications and prevents effective use of paper or screen area) are listed in [109].

Therefore, the authors of [109–112] in developing a programming technology for built-in real time algorithmic systems to overcome the disadvantages of the Harel model (absence of generation of finite codes) have combined the merits of SDL and Harel diagrams retaining alternative notation for each of them in designing an object behavior model. This is also true of the design of object-oriented software for real time systems [113]: the behavior model uses a state diagram with modified Harel notation and the SDL-diagram is used for improvement [114]. Therefore, the models of [109, 114], have specific and restricted application to logic control of technological processes. Leading companies do not use SDL diagrams in designing such logic controls.

Far more diverse is the behavior model elaborated in Unified Modeling Language (UML) [12, 91, 92, 115, 116]. It, in the opinion of its authors, is a collection of the best engineering approaches

used in object-oriented modeling for general purpose complex models. This language is based on the approaches of [11, 84, 117, 118].

In this language, the behavior of objects is described by a state machine, which, depending on the priority of states or activities, is defined by a state diagram or activity diagram. This is not correct, because transition parallelism is admissible in an activity diagram. Moreover, the state diagram is not a classical diagram and is regarded, though somewhat distinct [92], as an equivalent Harel diagram [90].

Since UML contains activity diagrams (an unexpected property [12]) that are not finite automata of automaton theory, there is some uncertainty in the choice of notation for describing the behavior of objects. Though these diagrams strongly resemble Grafset diagrams, which have been in use for the last twenty years in programming for automation of systems, they are worse since they do not contain "pointers" for transition conditions, but contain conditional vertices typical of algorithm schemes. While the vertices in a Grafset diagram are called stages, analogous vertices in an activity diagram are erroneously called states.

Moreover, as shown in [15] but not in [12, 92], every activity diagram (like every Grafset diagram) can be replaced by a system of interconnected transition graphs, which for a Grafset diagram can be realized by fewer number of internal variables.

The uncertainty in the choice of notation for describing the behavior of objects in UML steeply increases due to the fact that activity diagrams are regarded as constructs similar to SDL-diagrams. This corroborates the weak principles of formation of the "collection" under consideration.

I-Logix, one of the designers of UML, used only the Harel diagram for creating object-oriented built-in real-time systems [99], but, did not investigate the problem of verification of the behavior of these diagrams.

Harel diagrams are used in [119] for visualization of program components described by finite automata and intended for different platforms. In [181], only these diagrams are used for describing the behavior of objects.

The Harel notation is generalized in [120]. While transitions in Harel diagrams admit only conditional vertices inherent in stateless flow diagrams, these vertices may form more complex configurations [120]. A new term "stateflow" was thus coined to explain the close relationship between state and flow diagrams. Since these diagrams can be implemented as executable specifications in MATLAB [121], they can also be implemented in C and C++.

Since the Harel notation is highly specific, for low-level logic control systems (especially for PL controllers), far more convenient in algorithmization is a system of interconnected transition graphs containing a minimal number of components and notation ("never complicate unless unavoidable" (Okkam)) and based on classical automaton models helpful in formal and isomorphic programming even manually.

For this class of systems, the proposed technology is helpful in constructing a system of Boolean formulas for mathematical models used in designing programs with binary state coding. Such a possibility is nonexistent in other approaches.

In this technology, software and documentation for systems are designed and implemented.

Documentation is a vital problem; well-designed documents contain texts and description of programs and user instructions [122], whereas badly-designed documents omit program texts [123].

Finite automata, which are considered in [182] as modifications of processes, are presently used in controlling automatic switches of marine electrical equipment [124], designing nonprocedural programming languages for automatic control systems of technological processes [125], searching subrows [126], describing the behavior of objects [15, 127] in program realization of complex systems [87, 128–147], including "multifilar" controllers [30], and in other problems involving parallel

processes (five philosophers and forks [148], balls in window [149], synchronization of a chain of arrows [150]).

Transition graphs are the main tool for describing control processes in designing data processing programs by the methods of structural system analysis and design [40]. These graphs are also used for IDEF documentation (ICAM, DEFinition) and object transformation in technological processes [74].

The role and state-of-the-art of finite automata, transition graphs, and switch-constructs in programming are described in [151].

Moreover, transition graphs are used for describing the behavior of Turing machines used in the formal definition of algorithm [101, 152]. Therefore, this machine can be realized by switch constructs containing actions imitating the movement of "read–write" head. Therefore, the described approach can also used for a wide class of problems concerning the use of extended finite automaton model [38, 102]. This approach is used in object-oriented programming to define an object as an element that can be acted upon by changing its state.

At present, the automaton approach is in the incipient stage of application in program realization of algorithms. The "extinction" of automaton theory asserted in [153] is, in my opinion, strongly exaggerated. The reviewer of [8] defines a finite automaton as a tool for fighting against the monopoly of giant software designers [123] and even a hymn on the use of automata in programming is given in [50, 86].

In this connection, let us recall that K. Thomson, the designer of Unix [154], in reply to a question noted that a finite-state language generation machine was created, because a real selector telephone talk was a group of interacting finite-state machines. This language is used by Bell Labs for the main purpose—creation of such machines. Moreover, it is also used in driver design.

5. ALGORITHMIZATION AND PROGRAMMING OF "REACTIVE" SYSTEMS

The technology described in Section 2 is designed for creating algorithms and programs for logic control systems, in which input variables are fed only upon inquiry [155, 156] and cyclically [91]. Furthermore, programs produced are executed as compliers: switch constructs or their analogs are constructed from transition graphs and implemented directly and are, therefore, "active."

Grabovskii (Kaskog, Inc., St. Petersburg) applied the automaton approach [8] for program implementation of the logic part of control systems of Siemens controllers employing an advanced interruption mechanism. Tabulated transition graphs are used for automatically generating an array (resembling switch constructs), which is "passive" and processed by an interpreter with regard for inquiries when the microcontroller is implementing a transition to a graphs.

Teren'tev (Avrora, Inc.) used this approach to realize protocols in distributed control systems based on microcontrollers. The programs were complied formally and isomorphically from transition graphs.

Tunkel' (Avrora, Inc.) and I applied switch-technology in developing a generator control system based on an industrial computer and QNX platform, in which the control program is implemented as one process and the object-modeling program as another process. The switch-technology was slightly modified for reactive systems [39], because event-based control systems differ from systems based only on "cyclic controllers" [91]. Such systems are generally designed with industrial computers with real-time operating systems.

The modified switch-technology has the following features.

The basic concept is automaton, but not class, object, algorithm, or agent, as in other approaches. Programming can be called the automaton-oriented programming.

Automata are regarded not as isolated machines, but as components of an interconnected automaton system, whose behavior is formalized by a system of interconnected transition graphs.

The main model is a model of combined automaton, whose behavior is described by a transition graph containing only "simple" states (no hyperstates).

Notation for transition graphs (for embedded automata) is wider (than the notation of [8]).

The object domain is defined by technical specifications, which for automation of technological processes are usually defined by the customer in verbal form as a set of scenarios. In UML [91], a structural scheme is designed to determine control organization, equipment, and object interface.

Technical specifications are analyzed to determine the main functions, called automata (pump control automaton, temperature controller).

The states of each automaton are initially determined from the states of the object or part of object. If the number of states is large, they are determined through a control algorithm constructed with different notation (as an algorithm scheme [51]). Automata may also contain states to define operator faulty actions. Every automaton, if necessary, can be decomposed. Interactive analysis can be repeated and generates a list of automata and a list of states for each automata.

Unlike in traditional programming, the design includes a substage for coding the states of an automaton by a multivalued code, for example, decimal number of states.

Automata interact by exchanging the numbers of states, embedding depths, and calls. States may be embedded and addressed concurrently. An automata interaction scheme is constructed to determine interaction types. It formalizes the a system of interacting automata. This scheme replaces the object chart and partially the interaction chart (cooperation chart) used in object-oriented modeling [91].

Inputs are subdivided into transient events and input variables are fed in response to inquiries. Inputs are realized as input variables and events are used to reduce the system response time. Every input can be represented as an event or input variable.

Interruptions are processed by the operating system and transferred to the program as events and are processed as events by the corresponding handlers.

Certain input variables may be formed via comparison of input analog signals.

The numbers of states of other automata with which automata interact by exchanging the numbers of states are also regarded as inputs. Every input is an action, but not an activity. Groups of inputs and outputs are related to automaton. The relation of every automaton with its "environment" is formalized by a connection scheme describing the automaton interface. This scheme shows data sources and receivers, names of all actions and their notation, the automaton containing this scheme, and the automata in the scheme.

The names of automata, events, input variables, automaton state variables, and outputs begin with the letter A, e, x, y, and z, respectively. The number of corresponding automaton or action comes after the leading letter.

A system of interconnected automata forms a system-independent part (for example, independent of the operating system) of the program realizing the control system operation algorithm. Input variables, event handlers, outputs, auxiliary modules, user interfaces are realized by the system-dependent part of the program.

Event handlers contain calls to transition graph (automaton) functions with subsequent transfer to the corresponding events. Input and output functions are called by automaton functions. The functions forming auxiliary modules are called by input and output functions. Thus, automata exist at the "center" of the program.

If the system-dependent part of the program is designed as an automata (as an automaton for operator action file), then it can also be incorporated into the automaton interaction scheme.

If the platform is not changed, then auxiliary modules can be repeatedly used as samples [91].

Automata can be started both from the program system-dependent (event handlers) and system-independent parts. Embedded automata are sequentially started by transferring the "current" event according to routes in the automaton interaction scheme defining the states of automata at the startup of the main automaton. The automata startup and termination sequences are stored in depth search algorithm [157]. Automata are initiated by outputs with subsequent transfer of respective "internal" events. Automata can be started once with transfer of some event or repeatedly (in a cycle) with transfer of the same event.

The system is implemented such that automata-realizing functions cannot restarted until the completion of their work. Every automaton upon startup executes only one transition. After processing a routine event, every automaton preserves its state and remains idle until the next event appears.

Arcs and loops of transition graphs are labeled by arbitrary logic formulas, which may contain input variables and predicates verifying the numbers of states of other automata and the numbers of events. Along with transition conditions, arcs and loops may also contain a list of sequential outputs. Vertices are always stable and contain loops. If outputs are executed in a loop, then the loop is omitted. Otherwise, one or several loops are displayed, each of which is labeled, at least, by outputs. A vertex may contain a list of sequentially started embedded automata and a list of sequentially executed outputs. Vertices can be combined into groups to generalize "identical" incoming arcs at each transition graph. Incoming arcs of a vertex can also be combined into a line.

Every transition graph is tested for attainability, consistency, completeness, and absence of generating circuits.

This stage ends with the construction of a transition graph for every automaton in a system of interconnected automata.

A program for implementing transition graphs, input variables, event handlers, and outputs as functions is generated at the realization stage. It may contain submodules (for example, timer control module).

One internal variable is used to store the numbers of states (to distinguish states) of an automaton. A second auxiliary variable is used to distinguish the variations of a state.

A universal algorithm realizes the hierarchy of transition graphs with arbitrary embedding level.

Every transition graph is formally and isomorphically implemented by a separate (program) function generated from a template containing two switch constructs and one IF operator. The first switch initiates embedded automata and arc and loop actions. The IF operator verifies whether a state has been changed, and if changed, the second switch activates embedded automata and executes the actions at the new vertex.

After transition graphs have been generated, the subprogram text is corrected to suppress repeated inquiries of input variables labeling outgoing arcs of a state. Thus, "risk" is reduced [8].

Every input variable and every output are also realized by a function. Therefore, the switch-technology can be applied to problems other than logic control problems as well.

The names of functions and variables used in realizing automata are the same as those used in connection schemes of automata and transition graphs. For example, the variable holding the number of the current event is denoted by e.

All functions realizing input variables are stored in increasing order of serial numbers in one file and output-realizing functions in another file.

Functions realizing automata, input variables, and outputs contain calls to protocolling functions. This stage ends with the construction of a program structure scheme showing the interaction

between program parts. It may also contain an automaton interaction scheme, which is not separately generated.

The values of state variables of all automata are displayed on one screen during adjustments.

Automatic protocolling is implemented in the certification stage via inputs and outputs of automata functions that call protocolling functions. The protocol contains data on events, automata startup sequence, startup status of automata, transitions to new states, termination of automata operation, values of input variables and their outputs, and implementation time of each output. Along with a "complete" protocol, a "brief" protocol containing only events and their outputs is also generated.

Automata startup and stop messages in the complete protocol act as logical brackets for different embedding levels.

In the documentation stage, the results of software design and development are generated and stored in a log file. It contains the system structure scheme, program scheme, copies of user interface screens, a list of events, input variables, outputs, an automata interaction chart, description of notation used in transition graphs, templates for transition graphs of combined automata of arbitrary embedding level, verbal description (a fragment of technical specifications) for every automation as comments, automata connection scheme, a transition graph, the text of the function realizing the automaton, description of algorithms as a transition graph, texts of auxiliary modules and functions implementing input variables, event handlers, and outputs, program certification protocols in the form of examples, [158], programmer's instructions, and user's manual.

This technology can also be used in designing a model for the controlled object, for which a similar set of documents must be drafted.

If programs are modified later, the whole set of documents must be corrected [159] by listing all program modules and their cyclic checksums. Documentation controller must know the checksum of the initial file and all modified documents must be archived.

The advantages of the modified technology are the following.

Unlike in object modeling [91, 113], main modules are designed via automaton technology and only one type of a dynamic model—a system of interconnected transition graphs—is used.

This technology is effective in describing and implementing large-dimensional problems. Since transition graphs are used as a logarithm specification language, the program behavior is comprehensible and corrections are easily introduced both in specifications and in their implementations.

Joint use of connection scheme of an automaton with its transition graph aids in understanding the graph and joint study of this graph with its isomorphic subprogram aids in understanding the subprogram.

Documents of the software project admits corrections at any time.

The program is subdivided into system-dependent and system-independent parts without the use of any object-oriented approach.

In designing the system-independent part, the details of input and output implementations are hidden. They are displayed only in implementing the system-dependent part. The system-independent part of the program is designed and implemented separately.

Implementation of input variables and outputs as functions admits protocolling, easy transformation of one type of data sources and receivers to another type, and current program setup [158] at any time after the startup of the system-dependent part. Ordered storage of functions implementing input variables and outputs simplifies their correction.

Since only one internal variable is used for coding the automaton states, the automaton behavior can be observed by "tracking" the changes in the values of this variable. For a system of N

automata, the values of N multivalued variables are "tracked" by displaying the values of each variable on an "adjustment" screen determined by the automata interaction scheme.

Every transition graph is formally and isomorphically implemented by a template and, if necessary, the graph can be uniquely restored by this submodule.

The system-independent of the program is regular, readable, and correctable. It depends only on the compiler or interpreter of the program language of the platform. Only the system-dependent part needs modification in case of hardware replacement or operating system.

Since protocol is automatically introduced in terms of specifications, the compatibility of the program with the behavior of interconnected transition graphs for the events of input variables can be verified. This is implemented via comparison of the complete protocol with specifications. The program as a whole is verified with complete protocols. Brief protocols are used in verification tests and are also used as fragments for testing the system. Verification is usually associated only with the "algorithm–program" concept.

The brief protocol is used in determining output errors and the complete protocol is used in determining faulty automata. Therefore, they can be called "verification" and "diagnostic" protocols, respectively.

Since complete protocols are automatically generated in terms of automata, the system of interconnected transitions used in automaton specifications is not a "chart," but a mathematical model.

Though verification with these protocols is rather unwieldy, this method is more practical than other approaches to designing high-quality programs [160, 161].

A protocol or part of it generated by certain events corresponds to the scenario. Thus, the scenario [64] is designed automatically during program analysis, but not manually during program synthesis, as in other approaches [91, 113]. Manual construction of scenarios and formal synthesis of system-independent program from these scenarios for complex logic problems is virtually impossible.

The proposed approach does not exclude the possibility of interactive adjustments and certification.

The behaviour of the interconnected automata system is a modified form of the collective behavior of automata [162] and can be used in designing "multiagent systems containing reactive agents" [163].

The diesel generator control system designed by the modified technology has corroborated Brooks's opinion [158] that more time is required to design an algorithm for generating charts but not models than to write the system-independent part of the program isomorphic to the algorithm and Voas's opinion [164] that the CASE-tools, though they are fast, may generate incorrect codes from incorrect charts.

The full text of the submodule constructed by the modified technology and its complete protocol for verifying all transitions of the automaton realizing the toolbar control algorithm is given in [54].

6. FINITE AUTOMATA FOR PROGRAMMING PROGRAMMABLE LOGIC CIRCUITS

An approach similar to algorithm programming is widely applied in adjusting programmable logic circuits [165], also called programming [166]. Hardware description languages (HDL), in particular, are capable of converting program texts into functional schemes and vice versa. The description written in an HDL is compiled into a scheme [167]. The languages VHDL [168] and AHDL [169] are well-known languages of the HDL class.

In AHDL, conditional logic is implemented by "if then" and "case" constructs, of which the latter is an analog of the C switch construct. In [169], it is recommended that the "case" construct

must be preferred to the "if then" construct, whenever possible. This recommendation is used in implementing sequential logic described in terms of states with a state machine. An automaton with multivalued state coding (this term is not used in [168]) is defined as a state chart. State machines with synchronous outputs corresponding to Moore machines and state machines with asynchronous outputs corresponding to Mealy machines are illustrated with examples in [169).

Modification of the automaton approach for programming PL circuits consists of visualization state charts via "State CAD" packages [170].

Thus, the algorithm and software design technologies described in Sections 2 and 5 supplement the well-known approach to programming PL circuits and thus aid in hardware [171] and program implementation of logic control algorithms as in hardware and software co-design technology, in which finite automata are used as a high-level abstraction language [172].

7. APPLICATION OF THE STATE CONCEPT IN PROGRAMMING: TURING LAUREATES

I became aware of [173] and [174] after the publication of my book [8]. The technology based on finite automaton paradigm—one of the programming paradigms discussed by Floig [173]—has certain features described by Turning laureates, Turing himself [183], and von Neumann [184].

In 1966, Perlis [173] suggested that the description of language, medium, and computation rules must include states that are monitored in the course of implementation in order to facilitate nondestructive diagnosis of programs. By monitoring, he meant distributed control, which today is known as object-oriented programming.

In the same year, Dexter [174] introduced state variables via integer variables for describing the states of a system at any instant. He investigated the states that must be introduced, the number of values of a state variable, and what do these values denote. He determined a set of suitable states and then only constructed a program. He also compared a process with a state variable and connected processes through these variables. In his opinion, state charts might serve as a powerful program verification tool. All these support his idea that every program must be compiled correctly right from the beginning, but not debugged until it runs correctly.

In 1977, Becus [173] noted that the semantics were closely interwoven with state-state transitions in programming languages for Neumann computers and their disadvantage was computations changed the state. Consequently, every detail of every property must be nested in a state and its transition rule. He also noted that computer systems must be historically sensitive, but a system cannot be historically sensitive (admit the influence of one program on the behavior of the succeeding program) unless it has a state which the first program could change and the second program could accept it. Therefore, a historically sensitive model of a computing system must possess a state-changing semantics. He believed that a state might be considered as a whole and it must be changed only after "large" computations.

8. CONCLUSIONS

The automaton approach based on behaviorism, cognitive [57], and Gestalt psychologies [41] may enhance the design and quality of logic control programs for technological objects via autoprogramming, in which algorithmization and programming are implemented by man, because the notation system is a natural extension of the way of thinking, but not an alien formalism [173]). Transition graphs as an algorithmization language, like any language in which the solutions of a problem are written, may directly affect the progress of man's thoughts, compelling him to view the problem from a particular angle [37] defining the "thinking discipline" [20]. Transition graphs

are believed to be best for describing real processes [77], supplementing the objective process by a natural method of cognition of reality [175].

While procedural, object, and event programming techniques are based on choice, action, objects, and event, respectively [175], the automaton programming is based on automata, including inputs and outputs [8].

According to Schneider Electric (Kiev), Only 4% of Ukrainian users of its PL controllers use the Grafset language (IEC 1131-3) due to thinking inertia of programmers, who in most cases automatically transfer the style of PC programming languages to PL controller programming [9].

This situation, in my opinion, is a result of not only personal preferences, but also methodological specifics of this language and its implementation. For example, the Grafset diagram constructed for a logic control problem in [176] is a "chart," but not a model. This excludes the possibility of formal and isomorphic transformation of the chart into a program and use of the chart as a certifying test. Surprisingly, this chart is not used in [176] for writing programs in Grafset, but its "patterns" are used in heuristically constructing ladder schemes and programs in instruction language. Since transition graphs in logic control systems usually have multiple returns [8], their Grafset diagrams are nondemonstrative. The large number of binary variables used in PL controllers for vertex coding restrict the admissible number of vertices in transition graphs.

In the automaton technology, programming for PL controllers is best written in a structured text language [1], especially for PL controllers, containing a switch-like construct of C.

At present, the switch-technology is effectively used in designing control systems for three types of computing devices enumerated at the beginning of the review.

For logic control problems, the switch-technology is useful in constructing "quality" [177] programs, thereby corroborating the belief "what is not specified formally, cannot be verified and what is not verifiable cannot be faultless" [38].

Complexity is the reason for the difficulty in enumerating and understanding all possible states of a program—the root of its unreliability. Complexity is also the source of nonvisualizable states in which system protection is violated [158].

The switch-technology is based on predefined states and their visualizations. Therefore, it may be hoped, while exhibiting minimalism, at least for logic control and reactive systems [178], to be an approximation to a "silver pen" [158] for writing quality programs. In [178], Brooks speaks favorably about the Harel approach.

This technology can be regarded as a way of overcoming the "linear thinking crisis" [179] and another way is the use of hypertext.

The switch-technology is essential for meeting the IEC 880 software recommendations for the computers used in reliability systems of atomic power stations [180]. This Standard also specifies the design, correction, and control of software for control systems of atomic power plants. The use of the automaton paradigm in this technology as a central concept is consistent with control theory and this distinguishes it from the other programming paradigms.

REFERENCES

1. *International Standard IEC 1131-3: Programmable Controllers. Part 3. Programming Languages*, International Electrotechnical Commission, 1993.
2. *SIMATIC. Simatic S7/M7/C7. Programmable Controllers*, Catalog St 70, SIEMENS, 1996.
3. *TSX T607. Programming Terminal. User's Manual*, Telemecanique, 1987.
4. *Modicon Catalog and Specifier's Guide*, Modicon, AEG Schneider Automation, 1995.
5. *Programmable Controller. Melsec A. Programming Manual, Type ACPU, Common Instructions*, Mitsubishi Electric.

6. *ABB Procontic T200. Mid-Range Automation Systems using Modern Technology*, ASEA Brown Boveri, 1994.

7. *ET-PDS. Software for Programmable Logic Controllers*, Tosiba International (Europe), 1995.

8. Shalyto, A.A., *SWITCH-tekhnologiya. Algorimizatsiya i programmirovanie zadach logicheskogo upravleniya* (SWITCH-Technology. Algorithmization and Programming of Logic Control Problems), St. Petersburg: Nauka, 1998.

9. Guzik, I.M., *Standard IEC 1131: GRAFSET Languague—A Closer Look*, Schneider Automation Club, 1999, no. 6.

10. Lavrov, S.S., *Lektsii po teorii programmirovaniya. Uchebnoe posobie* (Lectures on Theory of Programming: A Textbook), St. Petersburg: S.-Peterburg. Gos. Univ., 1999.

11. Booch, G., *Object-Oriented Analysis and Design with Applications*, Redwood City: Benjamin/Cummings, 1994. Translated under the title *Ob"ektno-orientirovannyi analiz i proektirovanie s primerami prilozhenii na C++*, St. Petersburg: Nevskii Dialekt, 1998.

12. Fowler, M. and Scott, K., *UML Distilled: Applying the Standard Object Modelling Language*, Reading: Addison-Wesley, 1997. Translated under the title *UML v kratkon izlozhenii. Primenenie standartnogo yazyka ob"ektnogo modelirovaniya*, Moscow: Mir, 1999.

13. Klir, G.J., *An Approach to General Systems Theory*, New York: Van Nostrand, 1969. Translated under the title *Abstraktnoe ponyatie sistemy kak metodologicheskoe sredstvo. Issledovaniya po obshchei teorii sistem*, Moscow: Progress, 1969.

14. Kuznetsov, B.P., Structure and Complexity of Cyclic Program Modules, *Avtom. Telemekh.*, 1999, no. 2.

15. Odell, J.J., *Advanced Object-Oriented Analysis and Design using UML*, New York: SIGS Books, 1998.

16. Tal', A.A., Aizerman, M.A., Rozonoer, L.I., *et al.*, *Logika. Avtomaty. Algoritmy* (Logic, Automata, and Algorithms), Moscow: Fizmatgiz, 1963.

17. Shatyto, A.A., Application of Graph-Schemes of Algorithms and Transition Graphs in Program Realization of Logic Control Algorithms, *Avtom. Telemekh.*, 1996, nos. 6, 7.

18. Glushkov, V.M., *Sintez tsifrovykh avtomatov* (Synthesis of Digital Automata), Moscow: Fizmatgiz, 1962.

19. Yourdon, E. and Argila, C., *Case Studies in Object-Oriented Analysis and Design*, Upper Saddle River: Prentice Yall, 1996. Translated under the title *Strukturnye modeli v ob"ektno-orientirovannom analize i proektirovanii*, Moscow: Lori, 1999.

20. Ross, D.T., Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Trans. Software Eng.*, 1977, vol. SE-3, no. 1.

21. Chow, T.S., Testing Software Design Modeled by Finite State Machines, *IEEE Trans. Soft. Eng.*, 1978, no. 3.

22. King, D., *et al.*, On Object State Testing, in *18th Annu. Int. Computer Software & Applications Conf.*, Los Alamitos: IEEE Computer Society, 1993.

23. Turner, C.D. and Robson, D.J., The State-Based Testing of Object-Oriented Programs, in *Conf. Software Maintenance*, Los Alamitos: IEEE Computer Society, 1993.

24. Jorgenson, P. and Erickson, C., Object-Oriented Integration Testing, *Com. ACM*, 1994, no. 9.

25. Bunder, R.V., The FREE-Flow Graph: Implementation-Based Testing of Objects Using State-Determined Flows, in *8th Annu. Software Quality Week*, San Francisco: Software Research, 1995.

26. Burgonov, I.B., Kosachev, A.S., and Kulyamin, V.V., Use of Finite Automata in Program Testing, *Programmirovanie*, 2000, no. 2.

27. Kurshan, R.P., *Computer-Aided Verification of Coordinated Prosesses—The Automata-Theoretic Approach*, Princeton: Princeton Univ. Press, 1994.

28. Kurshan, R.P., Program Verification, *Notices ACM*, 2000, no. 5.

29. Shatyto, A.A., Implementation of Logic Control Algorithms in Functional Block Language, *Prom. ASU Kontrollery*, 2000, no. 4.

30. Martin, R.C., *Designing Object-Oriented C++ Applications Using the Booch Method*, Englewood Cliffs: Prentice Hall, 1995.

31. Prigozhin, I. and Stengers I., *Order in Chaos*, Moscow: Editorial URSS, 2000.

32. Traxtenbrot, B.A. and Bargzin, Ya.M., *Konechnye avtomaty. Povegenie i sintez* (Finite Automata: Behavior and Synthesis), Moscow: Nauka, 1970.

33. Gill, A., *Introduction to the Theory of Finite-State Machines*, New York: McGraw-Hill, 1962. Translated under the title *Vvedenie v meoriyu konechnykh avtomatov*, Moscow: Nauka, 1966.

34. Friedman, A.D. and Menon, P.R., *Theory and Design of Switching Circuits*, New York: Computer Sci., 1975. Translated under the title *Teoriya i proektirovanie pereklyuchatel'nykh skhem*, Moscow: Mir, 1978.

35. Kaner, C., Falk, J., and Hung Quoc Nguyen, *Testing Computer Software*, New York: Van Nostrand, 1993. Translated under the title *Testirovanie programmnogo obespecheniya*, Kiev: DiaSoft, 2000.

36. Rudnev, V.V., A System of Interconnected Graphs and Modeling of Discrete Processes, *Avtom. Telemekh.*, 1984, no. 9.

37. Budd, T., *An Introduction to Object-Oriented Programming*, Reading: Addison-Wesley, 1991. Translated under the title *Ob'ektno-orientirovannoe programmirovanie v deistvii*, St. Petersburg: Piter, 1997.

38. Zaitsev, S.S., *Opisanie i realizatsiya protokolov setei EVM* (Computer Network Protocols: Description and Implementation), Moscow: Nauka, 1989.

39. Shatyto, A.A. and Tukkel', N.I., SWITCH-Technology—An Automatic Approach to Software Design for "Reactive" Systems, in *Int. Conf. Telematics*, St. Petersburg: Inst. Exact Mech. Optics, 2000.

40. Kalyanov, G.N., *CASE. Strukturnyi analiz* (*avtomatizatsiya i primenenie*) (CASE. Structural Analysis: Automation and Application), Moscow: Lori, 1996.

41. Schultz, D.P. and Schultz, S.E., *A History of Modern Psychology*, Fort Worth: Harcourt Brace College, 1996. Translated under the title *Istoriya sovremennoi psikhologii*, St. Petersburg: Evraziya, 1998.

42. *Functinal Description. Warm-Up and Prelubrication Logic. Generator Control Unit*, Severnaya Hull N431, Norcontrol, 1993.

43. *Project 15640. AS 21. DG 1. Control. AMIE. 95564.12M*, St. Petersburg: Avrora, 1991.

44. *Sistema upravleniya turbokompressornym agregatom "Larina." Tekhnicheskoe opisanie. Prilozhenie 1. AMIE. 421417.010 TO.01* (Control System for "Larin" Turbocompressor Unit. Technical Description. Appendix 1. AMIE. 421417.010 TO.01), St. Petersburg: Avrora, 1998.

45. *Autolog 32. Rukovodstvo pol'zovatelya* (User Guide), St. Petersburg: FF-Automation.

46. Baglyuk, Yu.V. and Shatyto, A.A., *Programmiruemye logicheskie kontrollery "Autolog." 124 primera programm na yazyke "ALPro," realizuyushchikh algoritmy logicheskogo upravleniya* (Programmable Logic Controllers "Autolog." 124 Examples of Programs in ALPro Realizing Logic Control Algorithms), St. Petersburg: FF-Automation, 1999.

47. *Translyator "CF→ALPro" dlya programmirovaniya kontrollerov tipa "Autolog." Rukovodstvo pol'zovatelya* (CT→ALPro Translator for Programming Autolog Controllers. User Guide, NZF.TR 2.1RP), St. Petersburg: FF-Automation, 1999.

48. Litov, D.V., Automaton or Turing Machine, *Mir PK*, 1999, no. 3.

49. Baranov, S.I., *Sintez mikroprogrammnykh avtomatov (graf-skhemy i avtomaty)* (Synthesis of Microprogram Automata: Graph-Schemes and Automata), Leningrad: Energiya, 1979.

50. Lyubchenko, V.S., We Choose or We Are Chosen... (Choice of an Algorithmic Model), *Mir PK*, 1999, no. 3.

51. Zatuliveter, Yu.S. and Khalatyan, T.G., *Sintez obshchikh algoritmov po demonstratsiyam chastnykh primerov (avtomatnaya model' obobshcheniya po primeram* (Synthesis of General Algorithms from Particular Demonstration Examples: Automaton Model for Generalization from Examples), Moscow: Inst. Probl. Upravlen., 1997.

52. Matveev, V.I., Windows CE—A New PLC Development Stage, *Prib. Sist. Upravlen.*, 1999, no. 3.

53. Tukkel', N.I. and Shatyto, A.A., Comparison of Event and Automaton Approaches to Logic Control Programming, in *Conf. Telematics'99*, St. Petersburg: LITMO, 1999.

54. Shatyto, A.A., *Logicheskoe upravlenie. Metody apparatnoi i programmnoi realizatsii algoritmov* (Logic Control: Hardware and Software Implementation of Algorithms), St. Petersburg: Nauka, 2000.

55. Tukel, N., Programming with Use of a Switch-Technology, *Euroxchange. Special Student's Issue, ISA*, 1999, no. 2.

56. Shalyto, A.A., Technology of Program Realization of Logic Control Algorithms as a Lifetime Enchancing Tool. Problems of Guaranteed Lifetimes of Vessels and Ships, in *Sci. Tech. Conf.*, St. Petersburg: Krylov NTO, 1992.

57. Shalyto, A.A., Cognitive Properties of Hierarchical Representations of Complex Logical Structures: Architectures for Semiotic Modeling and Situation Analysis in Large Complex Systems, in *10th IEEE Int. Symp. Intel. Control*, Monterey, California, 1995.

58. Shatyto, A.A. and Antipov, V.V., *Algorimizatsiya i programmirovanie zadach logicheskogo upravleniya tekhnicheskimi sredstvami* (Alorithmization and Programming for logic Control of Technical Tools), St. Petersburg: Morintekh, 1996.

59. Bagljuk, Y.V. and Shalyto, A.A., SWITCH-Technology. Algorithmic and Programming Methods for Logic Control of Ship Equipment, in *Int. Conf. Informatics and Control*, St. Petersburg, 1997, vol. 1.

60. Kuznetsov, O.P., Makarevskii, A.Ya., Markovskii, A.V., *et al.*, Yarus—A Description Language for the Operation of Complex Automata, *Avtom. Telemekh.*, 1972, nos. 6, 7.

61. Kuznetsov, O.P., Logic Automaton Graphs and Their Transformations, *Avtom. Telemekh.*, 1975, no. 9.

62. Kuznetsov, O.P., Shipilina, L.B., Grigoryan, A.K., *et al.*, Logic Programming Languages: Their Design and Computer Implementation (as Illustrated by Yarus-2 Language), *Avtom. Telemekh.*, 1985, no. 6.

63. Lyubashin, A., Industrial and Built-in Systems: "Standard" Design, *PC Week*, 2000, no. 15.

64. ADAM-5510/P31. Everything for PC-Aided Automation, *Advantech*, 1999, vol. 91.

65. *Programmable Controller Fundamentals*, New York: Allen-Bradley, 1985.

66. *Micro Mentor: Understanding and Applying Micro Programmable Controllers*, New York: Allen-Bradley, 1995.

67. Jones, C.T. and Bryan, L.A., *Programmable Controller Concepts and Applications*, Int. Programmable Controllers, 1983.

68. Gilbert, R.A. and Llewellyn, J.A., *Programmable Controllers—Practices and Concepts*, Indust. Training Corp., 1985.

69. Lloyd, M., Grafset—Graphical Functional Chart Programming for Programmable Controllers, *Measur. Control Mag.*, 1987, no. 9.

70. Bryan, L.A. and Bryan, E.A., *Programmable Controllers: Theory and Implementation*, Industrial Text Corp., 1988.

71. Webb, J.W. and Reis, R.A., *Programmable Logic Controllers: Principles and Applications*, Englewood Cliffs: Prentice Hall, 1995.

72. Wisnosky, D.E., *SoftLogic: Overcoming Funnel Vision*, Wizdom Controls, 1996.

73. Highes, T.A., *Programmable Controllers*, Instrument Society of America (ISA), 1997.

74. Vernikov, G., Organization Activicity Research Elements: Standard IDEF3, *READ.ME*, 2000, no. 2.

75. Shatyto, A.A., Program Implementation of Control Automata, *Sudostroit. Prom. Ser. Avtom. Telemekh.*, 1991, issue 13.

76. Bowen, J.P. and Hinchey, M.G., Ten Commandments of Formal Methods, *Mir PK*, 1997, nos. 9, 10.

77. *Series 90-70. State Logic Control System. User's Manual*, North Arizona: GE Fanuc Autom., 1998.

78. *Industrial Controllers*, Matsushita Automation Controls.

79. *Software for Programmable Controllers and Visualization*, Festo Cybernetic.

80. Martynyuk, V.V., Analysis of Transition Graphs of Operator Schemes, *Zh. Vychisl. Mat. Mat. Fiz.*, 1965, no. 2.

81. Karpov, Yu.G., *Osnovy postroeniya kompilyatorov. Uchebnoe pocobie* (Compiler Design Principles: A Textbook), Leningrad: Leningrad. Pedagog. Inst., 1982.

82. Kas'yanov, V.N. and Pottosin, I.V., *Metody postroeniya translyatorov* (Translator Design Methods), Novosibirsk: Nauka, 1986.

83. Beizer, B., *The Architecture and Engineering of Digital Computer Complexes*, New York: Plenum, 1971, vol. 1. Translated under the title *Arkhitektura vychislitel'nykh kompleksov*, Moscow: Mir, 1974.

84. Booch, G., *Object Oriented Design with Applications*, Redwood City: Benjamin/Cummings, 1991. Translated under the title *Ob"ektno-orientirovannoe proektirovanie s primerami*, Kiev: Dialektika, 1992.

85. Stroustrup, B., *The C++ Programming Language*, Reading: Addison-Wesley, 1986. Translated under the title *Yazyk programmirovaniya C++*, Moscow: Radio i Svyaz', 1991.

86. Lyubchenko, V.S., New Songs on the Main (Programmer's Strap), *Mir PK*, 1998, nos. 6, 7.

87. Shlaer, S. and Mellor, S., *Object Lifecycles: Modeling the World in State*, Englewood Cliffs: Prentice Hall, 1992. Translated under the title *Ob"ektno-orientirovannyi analiz: modelirovanie mira v sostoyaniyakh*, Kiev: Dialektika, 1993.

88. Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program.*, 1987, vol. 8.

89. Harel, D., *et al.*, STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. Eng.*, 1990, no. 4.

90. Douglass, B.P., *UML Statecharts*, Massachusetts: I-Logix, 1998.

91. Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language. User Guide*, Massachusetts: Addison-Wesley, 1998. Translated under the title *Yazyk UML. Rukovodstvo pol'zovatelya*, Moscow: DMK, 2000.

92. *Unified Modeling Language (UML), Version 1.0*, Califormia: Rational Software, 1997.

93. Harel, D. and Pnueli, A., On the Development of Reactive Systems, in *Logic and Model of Computer Systems*, Apt, K.R., Ed., New York: Springer-Verlag, 1985.

94. Harel, D., *et al.*, On Formal Semantics of Software, in *2nd IEEE Symp. Logic in Computer Science*, New York: IEEE Press, 1987.

95. Coleman, D., Hayes, E., and Bear, S., Introducing Objectcharts, or How to Use Statecharts in Object-Oriented Design, *IEEE Trans. Soft. Eng.*, 1992, no. 1.

96. Harel, D. and Naamad, A., The STATEMATE Semantics of Statecharts, *ACM Trans. Soft. Eng. Metodology*, 1996, no. 10.

97. Harel, D. and Gery, E., Executable Object Modeling with Statecharts, *Computer*, 1997, no. 7.

98. Harel, D. and Politi, M., *Modeling Reactive Systems with Statecharts*, New York: McGraw-Hill, 1998.

99. Douglass, B.P., *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Massachusetts: Addison-Wesley, 1998.

100. Douglass, B.P., *Doing Hard Time: Using Object-Oriented Programming and Software Patterns in Real-Time Applications*, Massachusetts: Addison-Wesley, 1998.

101. Karpov, Yu.G., *Teoriya algoritmov i avtomatov: Kurs lektsii* (Course in Theory of Algorithms and Automata), St. Petersburg: Nestor, 1998.

102. *International Standard ISO 9074: Information Processing Systems. Open Systems Interaction. ESTELLE: A Formal Description Technique Based on an Extended State Transition Model*, 1989.

103. Bochman, G.V., Finite State Description of Communication Protocols, *Comput. Network Protocol Suppl. Liege*, 1978, vol. 2.

104. Dantine, A., Protocol Representation with Finite-State Models, *IEEE Trans. Commun.*, 1980, no. 4.

105. Brand, D. and Zafiropulo, P., On Communicating Finite-State Machines, *J. ACM*, 1983, no. 2.

106. *CCITT Recomendation Z.100: CCITT Specification and Description Language (SDL), COM X-R 26*, Geneva: ITU General Secretariat, 1992.

107. Braek, R. and Haugen, F., *Engineering Real-Time Systems*, Englewood Cliffs: Prentice Hall, 1993.

108. Gold'shtein, B.S., *Signalizatsiya v setyakh svyazi* (Signaling in Communication Networs), Moscow: Radio i Svyaz', 1997.

109. Ivanov, A., Koznov, D., and Murashova, T., The Behavior Model RTST++, in *Zapiski seminara kafedry sistemnogo programmirovaniya. CASE-sredstva RTST++*, (Seminar Annals, Department of System Programming: CASE-Tools of RTST++), St. Petersburg: S.-Peterburg. Gos. Univ., 1998, issue 1.

110. Parfenov, V.V. and Terekhov, A.N., RTST-Technology of Real-Time Built-In System Programming, in *Sistemnaya Informatika*, (Systems Informatics) Novosibirsk: Nauka, 1997, vol. 5.

111. Terekhov, A.N., RTST-Technology of Real-Time Built-In System Programming, in *Zapiski seminara kafedry sistemnogo programmirovaniya. CASE-sredstva RTST++*, (Seminar Annals, Department of System Programming: CASE-Tools of RTST++), St. Petersburg: S.-Peterburg. Gos. Univ., 1998, issue 1.

112. Dolgov, P., Ivanov, A., Terekhov, A., *et al.*, An Object-Oriented Extension of RTST-Technology, in *Zapiski seminara kafedry sistemnogo programmirovaniya. CASE-sredstva RTST++*, (Seminar Annals, Department of System Programming: CASE-Tools of RTST++), St. Petersburg: S.-Peterburg. Gos. Univ., 1998, issue 1.

113. Terekhov, A.N., Romanovskii, K.Yu., Koznov, D.V., *et al.*, REAL: Methodology and CASE-Tool for Designing Information Systems and Real-Time System Software, *Programmirovanie*, 1999, no. 5.

114. Koznov, D.V., Finite Automaton—The Base of Visual Representation of Object Behavior, in *Ob"ektno-orientirovannoe vizual'noe modelirovanie* (Object-Oriented Visual Modeling), St. Petersburg: S.-Peterburg. Gos. Univ., 1999.

115. *Microsoft. Solution'99*, Microsoft, 1999, no. 7.

116. Cook, S. and Daniels, J., *Designing Object Systems. Object-Oriented Modeling with Syntropy*, Englewood Cliffs: Prentice Hall, 1994.

117. Rambaugh, J., Blaha, M., Premerlani, W., *et al.*, *Object-Oriented Modeling and Design*, Englewood Cliffs: Printice Hall, 1991.

118. Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Massachusetts: Addison-Wesley, 1992.

119. *xjCharts. Release 2.0. User's Manual*, Experimental Object Technologies, 1999.

120. *STATEFLOW for Use with Simmulink. User's Guide. Version 1*, Massachusetts: Math Works, 1998.

121. *MATLAB. The Language of Technical Computing. Version 5.2*, Massachusetts: Math Works, 1998.

122. Zyubin, V. E., Fifth Anniversary of the Standard IEC 1131-3. Resume and Forecasts, *Prib. Sist. Upravlen.*, 1999, no. 1.

123. Herr, R., A New Turn, *PC Magazine*, 1998, no. 10.

124. Gubanov, Yu.A., Zalmanov, S.Z., and Kurnetsov, B.P., Control of Automatic Switches for Marine Electroheating System, in *3 Int. Conf. Marine Intelligent Technology "Morintex-99,"* St. Pertersburg: Marintex, 1999, vol. 3.

125. Mendelevich, V.A., Nonprocedural Languages—A New Generation of Design Tools for ACS Technological Processes, *Prom. ASU Kontrollery*, 2000, no. 1.

126. Cormen, T.H., Leizerson, C.E., and Rivest, R.L., *Introduction to Algorithms*, Cambridge: MIT Press, 1990. Translated under the title *Algoritmy. Postroenie i analiz*, Moscow: MTsNTO, 1999.

127. Odell, J.J., Approaches to Finite-State Machine Modeling, *J. Obiect-Oriented Prog.*, 1995, no. 1.

128. Heizinger, T., Manna, Z., and Pnueli, A., *Timed Transition Systems*, Technical Report TR 92-1263, Dept. Computer Science, Cornell Univ., 1992.

129. Ward, P. and Mellor, S., *Structured Techniques for Real-Time Systems*, Englewood Cliffs: Prentice Hall, 1985.

130. Hatley, D., and Pirbhai, I., *Strategies for Real-Time System Specification*, New York: Dorset House, 1987.

131. Drysinsky, D., *Visual Programming. Better State. Product Overview. R-Active Concepts*, Cupertino, 1993.

132. Selic, B., An Efficient Object-Oriented Variation of Statecharts Formalism for Distributed Real-Time Systems, in *IFIP Conf. Hardware Description Languages and Their Applications, CHDL'93*, Ottawa, 1993.

133. Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling*, New York: Wiley, 1994.

134. Borshchev, A.V., Karpov, Y.G., and Roudakov, V.V., COVERS—A Tool for the Design of Real-Time Concurrent Systems. Parallel Computing Technologies, in *Lecture Note in Computer Science*, 1995, no. 964.

135. Booch, G. and Rambaugh, J., *Unified Method for Object-Oriented Development. Documentation Set. Version 0.8*, Rational Software, 1996.

136. Sonkin, V.L., Martinov, G.M., and Lyubimov, A.B., Dialog Interpretation in Windows Interface Control Systems, *Prib. Sist. Upravlen.*, 1998, no. 12.

137. Ran, A.S., Modeling States as Classes, in *Tools USA 94*, Singh, M. and Meyer, B., Eds., Englewood Cliffs: Prentice Hall, 1994.

138. Ran, A.S., Patterns of Events, in *Pattern Languages of Program Design*, Coplien, J.O. and Schmidt, D.C., Eds., Massachusetts: Addison-Wesley, 1995.

139. Ilgum, K., Kemmerer, R., and Porras, P., State Transition Analysis: A Rule-Based Intrusion Detection Approach, *IEEE Trans. Software Eng.*, 1995, no. 3.

140. Corbett, J.C., Evaluating Deadlock Detection Methods for Concurrent Software, *IEEE Trans. Software Eng.*, 1996, no. 3.

141. Alur, R., Henzinger, T., and Pei-Hsin Ho, Automatic Symbolic Verification of Embedded Systems, *IEEE Trans. Software Eng.*, 1996, no. 3.

142. Heimdahl Mats, P.E. and Leveson, N.G., Completeness and Consistency in Hierarchical State-based Requirements, *IEEE Trans. Software Eng.*, 1996, no. 6.

143. Ardis, M.A., *et al.*, A Framework for Evaluaing Specification Methods for Reactive Systems. Experience Report, *IEEE Trans. Software Eng.*, 1996, no. 6.

144. Corbett, J.C., Timing Analysis of ADA Tasking Programs, *IEEE Trans. Software Eng.*, 1996, no. 7.

145. Zave, P. and Jackson, M., Where Do Operations Come From? A Multiparadigm Specification Technique, *IEEE Trans. Software Eng.*, 1996, no. 7.

146. Coen-Porisini, A., Ghezzi, C., and Kemmerer, R., Specification of Real-Time Systems Using ASTRAL, *IEEE Trans. Software Eng.*, 1997, no. 9.

147. Avrynin, G.S., Corbett, J.C., and Dillon, L.K., Analyzing Partially Implemented Real-Time Systems, *IEEE Trans. Software Eng.*, 1998, no. 8.

148. Ptiper, K., Synchronous C++ for Interactive Applications, *Otkrytye Sist.*, 1999, no. 3.

149. Lyubchenko, V.S., Billiards with Microsoft Visual C++ 5.0, *Mir PK*, 1998, no. 1.

150. Lyubchenko, V.S., The Mayhill Problem for Microsoft Visual C++ 5.0 (Synchronization of Processes in Windows), *Mir PK*, 2000, no. 2.

151. La Mot, A., Ratcliff, D., Seminatore, M., *et al.*, *Sekrety programmirovaniya igr* (Secrets of Game Programming), St. Petersburg: Piter, 1995.

152. Sholomov, L.A., *Osnovy teorii diskretnykh logicheskikh i vychislitel'nykh ustroistv* (Elements of the Theory of Discrete logic and Computing Devices), Moscow: Nauka, 1980.

153. Kuznetsov, O.P., Nonclassical Paradigms in Artificial Intellegence, *Izv. Ross. Akad. Nauk, Teor. Sist. Upravlen.*, 1995, no. 3.

154. Cook, D., Urban, D., and Hamilton, S., Unix and not only Unix. An Interview with Ken Tompson, *Otkrytye Sist.*, 1999, no. 4.

155. Shatyto, A.A., SWITCH-Technology. Algorithmization and Programming for Logic Control Problems, *Prom. ASU Kontrol.*, 1999, no. 9.

156. Shatyto, A.A., SWITCH-Technology. Algorithmization and Programming for Logic Control Problems, in *Int. Conf. Control*, Moscow: Inst. Probl. Upravlen., 1999, vol. 3.

157. Goodman, S.R. and Hedetniemi, S.T., *Introduction to the Design and Analysis of Algorithms*, New York: McGraw-Hill, 1977. Translated under the title *Vvedenie v razrabotku i analiz algoritmov*, Moscow: Mir, 1981.

158. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*, Reading: Addison-Wesley, 1995. Translated under the title *Mificheskii cheloveko-mesyats ili kak sozdayutsya programmnye sistemy*, St. Petersburg: Simvol, 2000.

159. Lipaev, V.V., *Dokumentirovanie i upravlenie konfiguratsiei programmnykh sredstv. Metody i standarty* (Documentating and Controling the Configuaration of Program Tools: Methods and Standards), Moscow: Sinteg, 1998.

160. Nepomnyashchii, V.A. and Ryakin, O.M., *Prikladnye metody verifikatsii programm*, (Applied Program Verification Methods), Moscow: Radio i Svyaz', 1988.

161. Woodcock, J. and Davies, J., *Using Z-Specification. Refinement and Proof*, Oxford: Oxford Univ. Press, 1995.

162. Varshavskii, V.I., *Kollektivnoe povedenie avtomatov* (Collective Behavior of Automata), Moscow: Nauka, 1973.

163. Round Table: Paradigms of Artificial Intellect, *Novosti Iskussvessnogo Intellekta*, 1998, no. 3.

164. Voas, D., Program Quality: Eight Myths, *Otkrytye Sist.*, 1999, nos. 9, 10.

165. Antonov, A.P., Melekhin, V.F., and Filippov, A.S., *Obzor elementnoi bazy firmy ALTERA* (A Review of ALTERA Elemental Bases), St. Petersburg: EFO, 1997.

166. Wirth, N., *Digital Circuit Design*, New York: Springer-Verlag, 1995.

167. Wirth, N., Hardware Compilation: Translating Programs into Circuits, *Computer*, 1998. Translated in *Otkrytye Sist.*, 1998, nos. 4, 5.

168. Armstrong, J.R., *Chip-Level Modelling with VHDL*, Englewood Cliffs: Prentice Hall, 1989. Translated under the title *Modelirovanie tsifrovykh sistem na yazyke VHDL*, Moscow: Mir, 1992.

169. *MAX + PLUS II. AHDL. Version 6.0*, California: Altera, 1995.

170. *Xilinx Foundation. M.1.5*, California: Xilinx, 1997.

171. Clare, C.R., *Designing Logic Systems Using State Machines*, New York: McGraw-Hill, 1973.

172. *A Framework for Hardware-Software Co-Design of Embedded Systems*, Technical Report, Berkley: Comp. Sci. Dept., Univ. California, 1995.

173. *Lektsii laureatov premii Tyuringa za pervye dvadtsat' let 1966–1885* (Lectures of Turing Laureates in 1966–1985), Moscow: Mir, 1993.

174. Dijkstra, E.W., *A Discipline of Programming*, Englewood Cliffs: Prentice Hall, 1976. Translated under the title *Vzaimodeisvie posledovatel'nykh protsessov: Yazyki programmirovaniya*, Moscow: Mir, 1972.

175. Deitel, H.M. and Deitel, P.D., *C++ How to Programm*, Upper Saddle River: Prentice Hall, 1998. Translated under the title *Kak programmirovat' na C++*, Moscow: Binom, 1999.

176. *TSX Nano. PL7-07 Language Self-Instruction Manual*, Groupe Schneider, 1997.

177. Pottosin, I.V., Program Quality Criteria, in *Sistemnaya Informatika* (System Informatics), Novosibirsk: Nauka, 1998, vol. 6.

178. Gerr, R., Debugging Humanity, *PC Magazine*, 2000, no. 5.

179. Chernyak, L., XML, Sideview, *Otkrytye Sist.*, 2000, no. 4.

180. Astrov, V.V., Vsilenko, V.S., and Tot'meninov, L.V., *Voprosy sozdaniya integrirovannykh sistem upravleniya yadernymi energeticheskimi ustanovkami: Sistemy upravleniya i obrabotki informatsii* (Design of Integrated Control Systems for Nuclear Power Plants: Data Control and Processing Systems), St. Petersburg: Avrora, 2000, vol. 1.

181. Boggs, W. and Boggs, M., *Mastering UML with Rational Rose*, San Francisco: Sybex, 1999.

182. Romanovskii, I.V., *Diskretnyi analiz* (Discrete Analysis), St. Petersburg: Nevskii Dialekt, 2000.

183. Turing, A.M., Computing Machinery and Intelligence, *Mind*, 1950, vol. 49, pp. 433–460. Reprinted under the title *Can a Machine Think?* in *The World of Mathematics*, Newman J.R., Ed., New York: Simon & Schuster, 1956, pp. 2099–2123. Translated under the title *Mozhet li mashina myslit'?*, Saratov: Kolledzh, 1999.

184. Von Neumann, J., General and Logic Theory of Automata, in Turing, A., *Mozhet li mashina myslit'?* (Can a Machine Think?), Saratov: Kolledzh, 1999.

185. Dale, N.B., Weems, C., and Headington, M., *Programming in C++*, Boston: Jones and Bartlett, 2000.

This paper was recommended for publication by O.P. Kuznetsov, a member of the Editorial Board