ISSN 0361-7688, Programming and Computer Software, 2008, Vol. 34, No. 1, pp. 27–43. © Pleiades Publishing, Ltd., 2008.
Original Russian Text © E.V. Kuzmin, V.A. Sokolov, 2008, published in Programmirovanie, 2008, Vol. 34, No. 1.

Modeling, Specification, and Verification of Automaton Programs

E. V. Kuzmin and V. A. Sokolov

Demidov State University, ul. Sovetskaya 14, Yaroslavl, 150000 Russia
E-mail: egorkuz@uniyar.ac.ru, sokolov@uniyar.ac.ru

Received October 4, 2006

DOI: 10.1134/S0361768808010040

1. INTRODUCTION

The paper is devoted to the description, specification, and verification of models for program constructed by means of the automaton approach [1–5].

The automaton programming technology is a modern Russian development, which is actively studied and supported by a number of Russian research groups. In the automaton approach to the program design and construction, the program is divided into two—system-independent and system-dependent—parts. The former part implements logic of the program and is given by a system of the interacting Moore–Mealy automata. The design of each automaton consists in the creation of a link scheme describing its interface and a transition graph determining its behavior by a verbal description of the desired automaton (declaration of purposes). Given these two documents, a program module corresponding to the automaton can formally and isomorphically be constructed (after which its system-dependent part can be implemented).

The automaton programming does not depend on the platform, operating system, or programming language. The automaton programming technology is very effective in constructing software for reactive systems and logical control systems. This technology, which does not exclude using other methods of "erra-free" software construction, is considerably more constructive than other methods, since it allows one to start to "fight against errors" as early as at the algorithmization stage.

It is interesting that all existing methods for correctness analysis (verification) of complex systems, such as simulation modeling, testing, deductive analysis, and model checking method, can successfully be applied to the automaton programs.

The simulation modeling and testing envisage carrying out tests before manufacturing the system. The simulation modeling is applied to an abstract schema or prototype, and the testing is used for the product itself. In the case of software, the simulation modeling and testing include supplying certain input data and observing appropriate output results. These methods allow one to find errors (they are especially efficient at early stages) but cannot guarantee finding all errors or estimate how many errors still left in the program.

Although the deductive analysis [6] is a very labor-consuming method with strong referencing to semantics of the programming language and cannot completely be automated and used without expert control, it still can be used for completed automaton programs for immediate checking of the correctness of the procedures (written in a high-level language) corresponding to output actions or input queries. Each output action performs its own, usually small, task, the correctness of the implementation of which can be checked. Thus, we can say that the automaton structure of the program advantages application of the deductive analysis. However, this method turns out useless for checking program logic.

On the other hand, the logic of automaton programs can conveniently be verified by the model checking method [7]. In the framework of this method, a formal finite model of the program is constructed, and the properties being checked are specified by means of temporal logic formulas. The satisfiability of the temporal formulas specifying model properties is checked automatically. It is important to note that the construction of the program model for automated verification is a very difficult task. When constructing the model for a program written in a traditional way, we arrive at the problem of adequacy of this program model and the original program. The model may not take into account a number of program features or generate nonexisting properties. From the standpoint of modeling and analysis of program systems, the automaton approach has a number of advantages compared to the traditional approach. In the automaton programming, the problem of model adequacy does not arise, since the set of the interacting automata describing the program logic is already an adequate model, by which the program module is constructed in a formal way. This is certainly a great advantage of the automaton technology. In addition, the model has a finite number of states, which is a

necessary condition for successful automatic verification in practice, since the checking of the model is a search method.

One of the most popular temporal logics for specification and verification of properties of program systems are the CTL (computation tree, or branching time, logic) or LTL (linear-time logic) logics. The use of the LTL for the verification of the automaton programs deserves special attention, since any formula in this logic is essentially a Buchi automaton describing (accepting) infinite admissible paths of the Kripke structure, which, in turn, specifies the behavior (all possible implementations) of the automaton model checked for correctness. This allows us to basically use only a simple concept of the "automaton" in the specification and verification of automaton programs.

Thus, the above-said allows us to speak of the possibility of efficient use of the model checking method for verification of the automaton programs (for analysis of correctness of the automaton program logic).

There exist a number of software tools for verification of abstract models constructed in the framework of certain formalisms (Petri nets, interacting asynchronous processes, real-time automata, hybrid automata, and the like) and based on general approaches and technologies. These tools of classical verification by means of the model checking method include, e.g., CPNTools [8], SPIN [9], SMV [10], and CADP [11], which use high-level colored Petri nets, the Promela language, asynchronous processes, and the LOTOS language, respectively, for the formal model specification languages. On the one hand, it is commonly accepted to consider that the goal of the creation of such program products is, basically, development of various theoretical and applied methods of model verification. On the other hand, many of these methods have specification languages designed for a particular field, for example, modeling and analysis of communication protocols or development of synchronous hardware logic circuits. However, there do not exist software verification tools designed directly for support of the automaton programming technology, which does not exclude (or even implies) active use of the available (time-proved) developments of the leading world's laboratories, which are usually accompanied for a long time by discussions at appropriate scientific conferences (which are often entirely devoted to these products). The existing verification tools suggest basically extracting of the model from the real program and, then, specifying this model in the context of certain abstract formalism. Then, properties of the constructed formal model are specified and checked. The behavior of this model may be, generally, quite different form that of the original program, in particular, because of specific features of the formalism used. The goal of the studies reported in this paper is the creation of a software complex intended for designing software products by means of the automaton models, from which, after appropriate correctness analysis, the desired programs are generated. Thus, we are going to analyze an adequate automaton model responsible for functional program logic. By the automaton model, functional requirements imposed on the software are checked (with respect to the specification). If, after checking the automaton model, errors are found in the program, they, most likely, will refer to the implementation details rather than to the functional program logic. Hence, the correction of these errors will not require global redesign of the entire program.

In this paper, we consider a hierarchical model of construction of automaton programs. For specification and verification of the automaton programs, a technology of application of the model checking method is proposed. The possibility of specification (in terms of simplicity and perception adequacy) of structural and semantic properties of automaton programs with the help of temporal logic is studied. Note that the emphasis is placed more on the semantics of the automaton programs rather than on their structure. This feature differs our method from the majority of others, where the interacting automata (with various extensions) are considered as certain abstract formalism for which, along with the decidability of classical (from the standpoint of theoretical information science) properties, only complexity of the model checking method related to the structure of the automata and their interaction scheme is estimated. Moreover, the choice of the observable behavior of the automaton program model in this paper is very important, since it greatly affects both classes (types) of the verified properties and the very possibility of adequate specification. On the whole, the paper is aimed at supporting the automaton approach to programming reactive systems and complex logical control systems.

2. HIERARCHICAL MODEL OF AUTOMATON PROGRAMS FOR REACTIVE SYSTEMS AND LOGICAL CONTROL SYSTEMS

Let us consider a hierarchical system of interacting deterministic finite automata given by

$$\mathrm{A} = (A_0, A_{11}, \ldots, A_{1k_1}, \ldots, A_{n1}, \ldots, A_{nk_n}),$$

where n and k_i $(1 \le i \le n)$ are positive integers. The automaton A_0 is called main, and the others are called nested automata. All automata are related through a hierarchy with respect to nesting. An automaton A_{ij} can transfer control to an automaton A_{i+1k} that occupies a lower level in the hierarchy. In this case, the automaton A_{ij} is said to be principal, and the automaton A_{i+1k}, nested. For each nested automaton, there exists only one principal automaton in which it is nested.

The automaton system A is considered to be a reactive control system for an object. The system A receives from the object events that characterize change of its states and asks the object about its current parameters, which is also considered to be an input action on A.

At the same time, the control system reacts to the arriving information and affects the control object.

In addition to the above-described interaction with the "environment," the automata interact with one another inside the system by transferring control from the principal automaton to the nested ones when certain events occur and watching their current states.

In the considered model, only the main automaton A_0 receives events from the control object and reacts to them; the nested automata can address the control objects only with a query on the states of its parameters (note that a nested automaton can send a query only if the principal automaton transferred control to it).

The above-described model of the interaction of the control system A and the control object is shown in Fig. 1. In the figure, the events are denoted by the letter **e**; the queries to the control object (input variables), by **x**; the current state of an automaton A_{ij} is stored in variable $\mathbf{y}_{ij}$; and the output actions are denoted as **z**.

The behavior of any automaton A of the system A is similar to the behavior of the whole system in the sense that the automaton A reacts to the occurring events and, depending on its state, states of the nested automata, and the state of the control object (i.e., values of the input variables), affects the control object or the nested automata by transferring them control when some event occurs.

Let us denote by $E_A = \{e_1, \ldots, e_k\}$ the set of all names of the events to which the automaton A reacts. Let us define an event variable e on the set E_A. In this variable, the name of the event occurring for the automaton A is placed. The current value of e is denoted as $E_A(e)$.

Let us introduce the set $X_A = \{x_1, \ldots, x_n\}$ of queries of the automaton A to the control object. Each query is considered as a certain predicate the truth of which depends on the state of the control object parameters.

Let also $Z_A = \{z_1, \ldots, z_r\}$ denote the set of output actions of A. Actions z_i are classified into two groups. The first group includes direct actions on the control object. The actions from the second group are control transfers to the nested automata occurring after some events (generation of events for nested automata). In this case, the output action z_i has the form $A'(e'_j)$, where A' is a nested automaton and e'_j is an event generated by the automaton A for the nested automaton A' to which the control is transferred for processing this event.

Then, the automaton A of the control system A can be represented as a set $(\Sigma, Q, q_0, E, X, Z, \delta)$, where

- $Q = \{q_0, q_1, \ldots, q_n\}$ is a finite set of states of A,
- q_0 is an initial state,
- $\Sigma = \{a_1, a_2, \ldots, a_k\}$ is a finite alphabet of labels of the transition arcs,
- $\delta : Q \times \Sigma \longrightarrow Q$ is a function of transitions from one state to another.

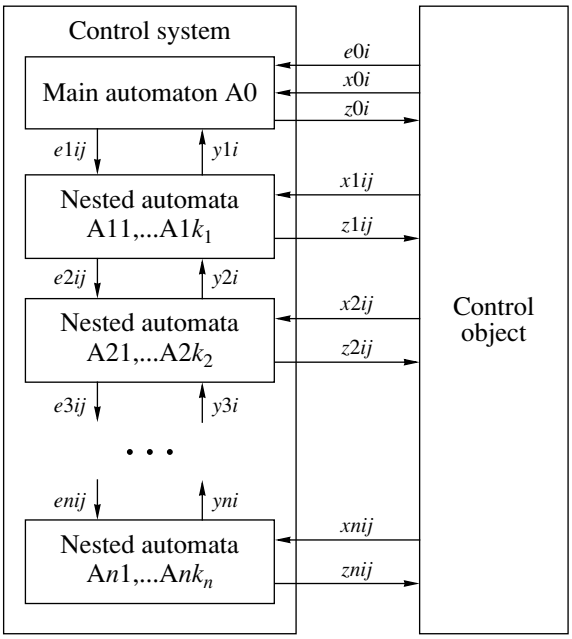

Fig. 1. The model of interaction of system A and the control object.

Each transition fires by a certain rule. Before describing the transition rules, we introduce some notation.

For a transition label $a \in \Sigma$, $E(a)$ denotes the event to which A reacts upon firing the transition with the label a.

Let $X(a)$ denote the set of queries to the control object the truth of which is required for firing the transition with the label a.

Let Z^* be a set of finite sequences of output actions. Then, for $a \in \Sigma$, $Z^*(a) \in Z^*$ denotes the sequence of the output actions that occur when the transition with the label a fires.

For an arbitrary state q of an automaton A, we introduce the notation $Z^*(q) \in Z^*$ for the sequence of output actions that are to be performed when the automaton comes to the state q.

Finally, let $Y(a)$ be a predicate depending on the states of the nested automata that must be true in order that the transition with the label a fire.

The rule of the transition from a state q to a state q' has the following form:

q, a: **if** $e = E(a)$ **and** $(\forall x \in X(a): x = true)$ **and** $Y(a) = true$

then $Z^*(a)$; $Z^*(q')$; **goto** q'.

It follows from the above-said that the rule of the transition to a new state in the general case can be described as follows. Having received an event, the automaton reacts (or does not react) to it (with reaction being determined by its current state), asks the control objects about its parameters (input variables), takes into

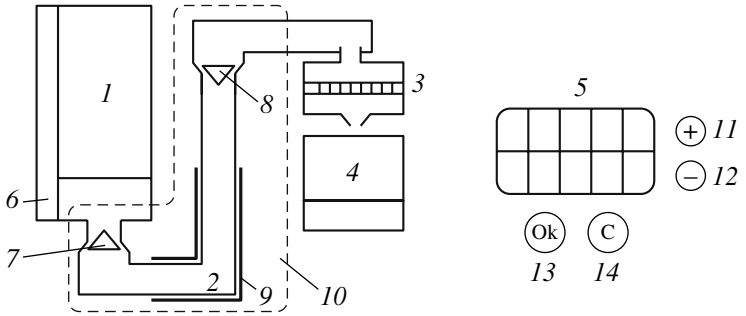

Fig. 2. Coffee-making machine: *1* – container for water, *2* – boiling container, *3* – coffee filter, *4* – flask for the coffee, *5* – display, *6* – water sensor, *7* – input valve (valve 1), *8* – output valve (valve 2), *9* – heater, *10* – boiler, *11* – "+" (increase) button, *12* – "–" (decrease) button, *13* – "ok" button, *14* – "C" (cancel) button.

account states of the nested automata and, then, performs a sequence of output actions, including the actions that are required to perform when it occurs in a new state. Only after this, it is transferred to the new state (which, in the case of a loop, may coincide with the old state).

An output action of the first kind, which is aimed at the control object, is considered to be performed immediately after the application. An output action of the second kind, which is a control transfer to a nested automaton in response to an incoming event, is considered to be performed only after the reaction of the nested automaton to this event. The latter reaction consists in the following: either the automaton transfers to the new state (one of the transitions of the nested automaton fires) or the event is ignored by the nested automaton (none of the transitions can fire). Until the output action of the second type is performed, the operation (the process of transition to the new state) of the principal automaton is postponed.

It is important to note that the transition rules (more specifically, transition firing conditions) must satisfy the determinacy condition (or orthogonality); i.e., not more than one transition can be ready to fire when an event occurs. If none of the transitions can fire in the current state when an event occurs (transition conditions are not fulfilled), then the event is ignored (the event variable e for this automaton is set equal zero).

Further, we consider a hierarchical model of automaton programs on the example of a coffee maker control system [12].

3. AUTOMATON MODEL OF A COFFEE MAKER CONTROL SYSTEM

We consider a system-independent part of the automaton program responsible for the *logic* of the coffee maker control.

The coffee maker model is shown in Fig. 2. It allows one to select the number of coffee portions by means of the "+" (increase the number of portions by one) and "–" (reduce the number of portions by one) buttons.

Provision is made for the indication of lack of water and basic faults (for example, heater failure or a defective valve). The number of portions varies from 1 through 5. An attempt to increase the maximum number of portions by pressing the "+" button does not result in any change (similarly, if the number of portions is 1, pressing the "–" button is ignored). A separate object is a boiler consisting of valves 1 and 2, heater, and container for boiling water. The coffee maker display is divided into panels used for indicating states of the valves, the heater, and the coffee-making machine itself.

There are three control subobjects (boiler, valves, and heater) in the coffee maker, each of which has its own control subsystem. Each control subsystem is represented as a hierarchy of finite Moore–Mealy automata. As a result, the logical part of the coffee maker control system has the form of a hierarchical system of interacting automata. An automaton occupying a higher place in a hierarchy controls the automata nested into it by generating events and transferring control to the nested automata when the events occur. In addition, the automaton watches states of the nested automata since its own transitions may depend on these states.

The automata interaction diagram is shown in Fig. 3. The main automaton $A0$ receives events $e0i$ from the control panel (button panel) and reacts to them. The special event $e0$ generated by the system timer is used for checking conditions of the transitions (conditions on the automaton arcs) that do not suggest reactions to any event $e0i$. The automaton $A0$ interacts with the automaton $A1$ by transferring control to it when events $e1i$ and $e0$ occur. The automaton $A0$ watches states of $A1$ through variable $y1$. In this interaction, $A0$ is considered to be a principal automaton, and $A1$, a nested automaton. The automaton $A1$ interacts with the nested automata $A2$, $A31$, and $A32$ in a similar way.

The automaton $A0$ implements logic of the coffee maker control. It reacts to pressing buttons on the control panel (when specifying the number of portions, starting boiling process, canceling coffee making, error reset, etc.) by transferring control to the boiler control

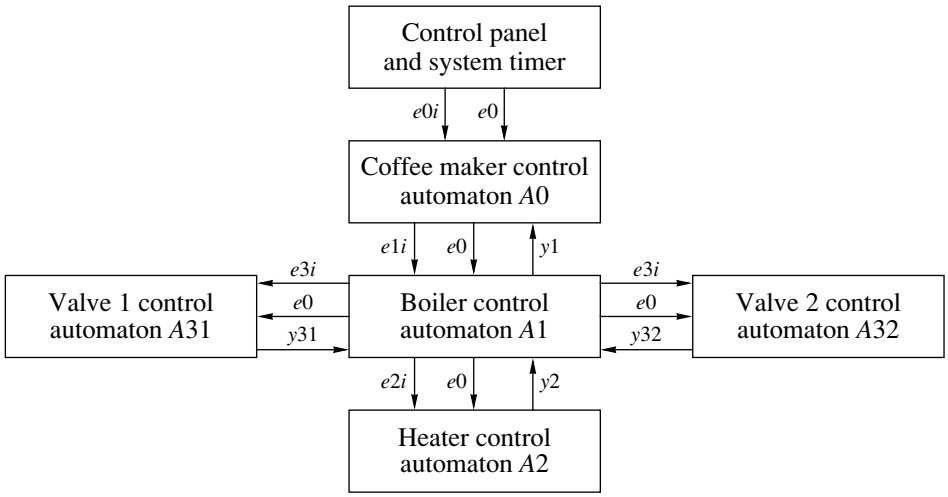

Fig. 3. The control automata interaction diagram.

automaton $A1$ for the time of boiling. The information on the current states of the coffee maker is depicted on the display.

The scheme of links and transition graph of the coffee maker control automaton $A0$ are shown in Fig. 4.

The automaton $A1$ implements logic of the boiler control, which is responsible for the coffee-boiling process.

In the process of boiling, the efficacy of the valves and heater and availability of water are checked. If one of the valves or heater is out of order, the boiling is terminated, and an appropriate warning is displayed. If the water is lacking in the container, the boiling is postponed until some water is added or the user interrupts the boiling process.

If no breakage occurs, the boiling process is repeated until the required number of coffee portions is done. One boiling loop corresponds to one portion.

The automaton $A1$ interacts with the control automaton $A0$ informing the latter about its current state and receiving an event from it. The automaton $A1$ interacts also with the automata $A31$, $A32$ (responsible for control of valves 1 and 2), and $A2$ (controlling the heater) by receiving information about their states and sending them events.

The scheme of links and transition graph of the boiler control automaton $A1$ are presented in Fig. 5.

The automaton $A2$ implements logic of the heater control. It supports heater temperature in a prescribed temperature range (minimum and maximum values of temperature are set). It is possible to determine whether there is a breakage of the type "heater does not work."

After turning the heater on, it continues to heat up to the lower boundary of the working range. If it does not reach the required temperature, the heater is considered out-of-service. The heater is turned on when the temperature reaches the lower boundary of the working range and is turned off when it reaches the upper boundary.

The automaton $A2$ interacts only with the boiler control automaton $A1$ and does not have nested automata.

The automata $A31$ and $A32$ are completely identical. Speaking of automaton $A3$, we mean both $A31$ and $A32$. The automaton $A3$ implements logic of the control of valve operation in the mode "open–pause–close." The automaton has four states and, having no nested automata, interacts only with the boiler control automaton $A1$ sending its state to the latter and receiving an event from it. The automaton $A3$ starts its operation loop by the event $e31$ ("open the valve"). First, the valve opens; then, it pauses to water or blow off steam; and, finally, it closes. If, during a certain time period, the valve does not open (or does not close), it is considered out of service.

The schemes of links and transition graphs of the heater control automaton $A2$ and valve control automaton $A3$ are presented in Fig. 6.

4. SPECIFICATION AND VERIFICATION OF AUTOMATON MODELS

When constructing an automaton program in the framework of a hierarchical model, the program logic is concentrated in the main automaton, which sends controls to the nested automata depending on the behavior of the controlled object. Each automaton of the program interacts only with the principal automaton and the nested automata, which facilitates understanding of the program. In the course of the design or verification of such an automaton program, it is possible to consider a part, or a subtree, of the automaton system, depending on what function is implemented by the considered system of automata. Any subsystem of interacting automata in a hierarchical model is a tree of automata, which

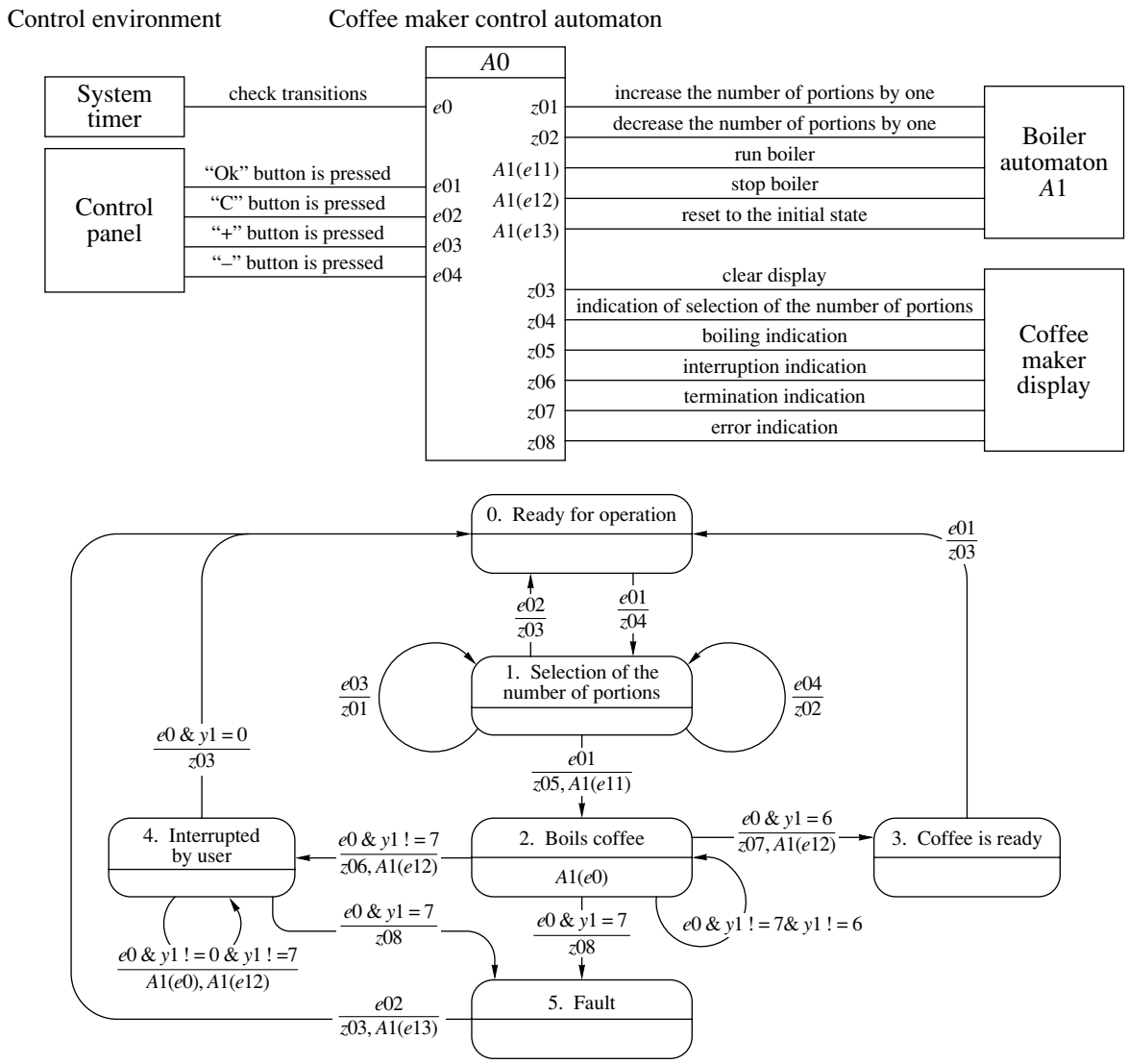

Fig. 4. The scheme of links and transition graph of the coffee maker control automaton *A*0.

can be considered as a separate system (as a separate automaton program). This makes it possible to change the scale of the system by considering the automata that are not relevant to the design at a given moment as a part of the environment, i.e., an external control object, and to focus only on the subsystem analyzed. Finally, specification and analysis of properties of an automaton program under verification are simplified if the structure of the verified model is easy to understand.

An automaton program model is an ideal object for automated verification by the model checking method [7], which consists in the following.

The behavior of a program model is described by a finite system of transitions, which is called Kripke structure. The finiteness of a transition system implies that the system has a finite number of states. Under some simplification, a finite transition system may be viewed as a finite directed graph with an explicitly separated initial vertex.

Further, in the framework of the Kripke structure, with the use of a temporal logic language, properties of the program model are specified, and the truth of these properties for this model is checked. Using simple objects such as a finite system of transitions and temporal logic formulas, it is possible to automatically verify model properties given in the form of formulas.

Temporal logics play an important role in the formal verification. They are used for expressing process properties, such as, for example, *"the process never reaches a deadlock state"* or, *"in any endless execution of a process, action b occurs an infinite number of times*."

The objective of the *model checking* is to determine whether a property given by a temporal logic formula is satisfiable for the process specified by a transition sys-

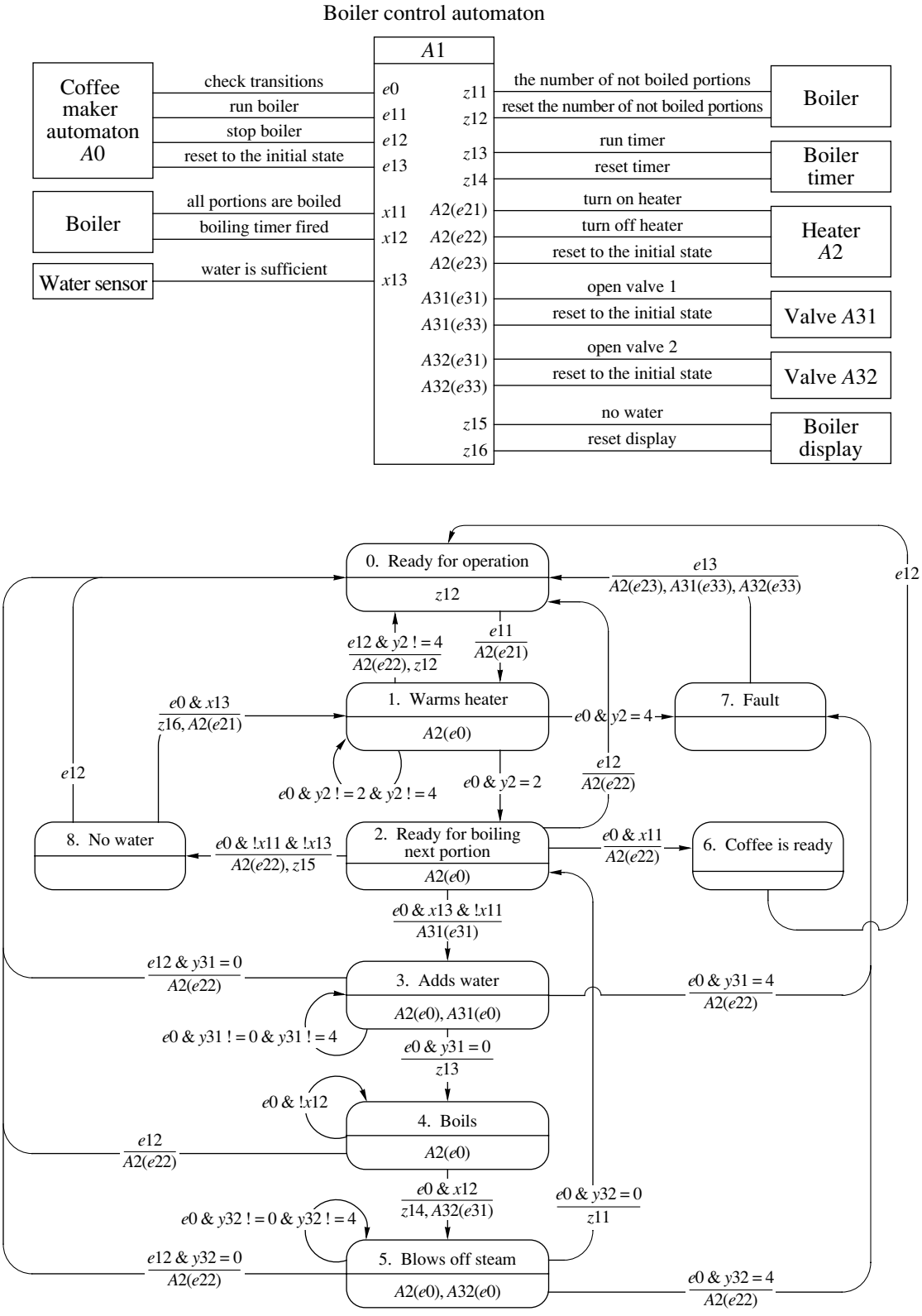

Fig. 5. The scheme of links and transition graph of the boiler control automaton $A1$.

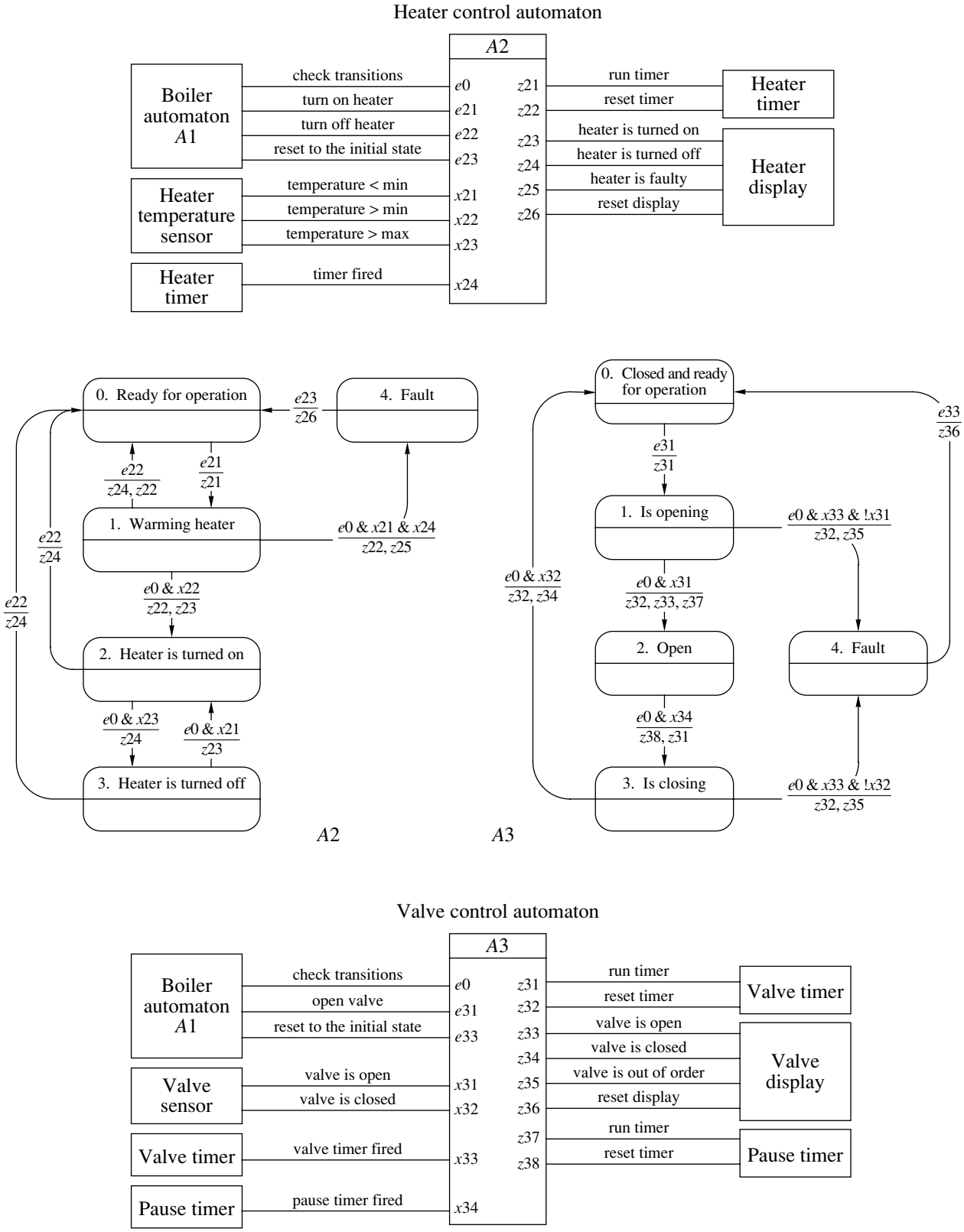

Fig. 6. The schemes of links and transition graphs of the heater automaton *A*2 and valve automaton *A*3.

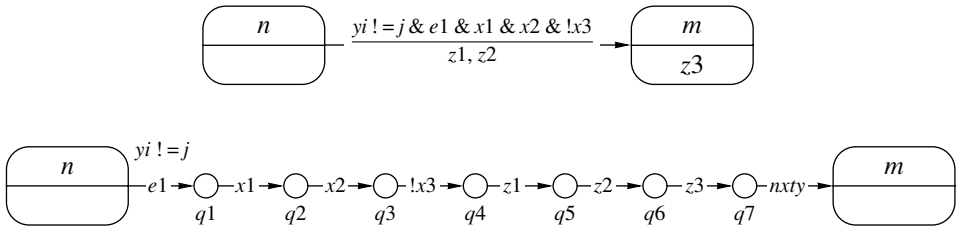

Fig. 7. Separation of intermediate states for an automaton transition.

tem (Kripke structure). Let us formulate the model checking problem.

Model checking problem

Given a finite transition system (Kripke structure), an initial state s_0 of this system, and a temporal logic formula φ, *find out* whether the state s_0 satisfies the temporal logic formula φ.

5. KRIPKE STRUCTURE OF AUTOMATON MODEL

The *Kripke structure* over a set of elementary propositions P is a transition system $\boldsymbol{S} = (S, s_0, \longrightarrow, L)$, where

• S is a finite set of states;

• $s_0 \in S$ is an initial state;

• $\longrightarrow \subseteq S \times S$ is a total transition relation (totality means that, for each state $s \in S$, there exists a state $s' \in S$ for which $(s, s') \in \longrightarrow$, i.e., $s \longrightarrow s'$);

• L: $S \longrightarrow 2^P$ is a function that marks each state by a set of elementary propositions that are true in this state.

A *path from a state* s_0 in the Kripke structure is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that, for all $i \geq 0$, $s_i \longrightarrow s_{i+1}$.

For a path $\pi = s_0 s_1 s_2 s_3 \ldots$, π^i will denote the suffix of π obtained by deleting the first i states from π; for example, $\pi^1 = s_1 s_2 s_3 \ldots$, and $\pi(i)$ will denote the ith state of the path, $\pi(0) = s_0$, $\pi(1) = s_1$, and so on.

For an arbitrary hierarchical program constructed on the basis of the automaton programming, we consider its automaton model A, which is a set of interacting automata $(A_0, A_1, \ldots, A_n)$.

For this automaton model A, we construct the Kripke structure, which, on the one hand, describes all possible states of system A and, on the other hand, specifies semantics of elementary propositions that are true in these states.

Consider an arbitrary automaton A_k from system A. In the automaton A_k, in addition to the basic states, we separate a set of its intermediate states, the automaton states during its transition from one state to another. An intermediate state of the automaton transition is fixed every time when the automaton performs one of the elementary actions, i.e., reacts to an event e_k, addresses the control object with a query of values of input variables x_k ($!x_k$), or produces an output action on the control object or nested automaton z_k.

Let us demonstrate the idea of separation of intermediate states for a transition of an arbitrary automaton A_k (Fig. 7).

In Fig. 7, the intermediate transition with the label *nxty* is introduced in order to have an opportunity to directly track the moment when the nested automaton transits to the next basic state with simultaneous control transfer to the principal automaton (in the case of the intermediate transition *nxty* of the main automaton, the main automaton remains to be active automaton). An internal transition with label $e1$ may occur when the event $e1$ occurs under the condition that $yi! = j$. All other transitions have no conditions and are active when the automaton comes to the corresponding intermediate states. In the sequence of transitions by internal states depicted in the figure, the first transition is necessarily that corresponding to the input event; then, transitions with labels of input queries come; and, finally, we have transitions marked by output actions. Thus, we can divide the sequence of internal states into three groups, which go one after another in accordance with the specified order. In each group, labels of internal transitions are placed in the order the corresponding elements are located in the expression on the arc of the basic (original) transition.

Further, for an arbitrary automaton, under a transition to the next state, we mean an *internal* transition to a basic or intermediate state.

To each automaton A_k from set A, we assign variables y_k, xm_k, zd_k, ev_k, and st_k ($0 \leq k \leq n$). For the entire system of automata A, we introduce variables *ev*, *xm*, *zd*, *act*, *auto*, and *evnt*.

The variables introduced have the following meaning:

(1) Variable y_k stores the last basic state of the automaton A_k.

(2) Variable xm_k contains the last executed query of control object parameters. The values of xm_k are names of input actions in the direct or inverse form; i.e., xm_k may contain either x or $!x$, where $x \in X_{Ak}$.

(3) Variable zd_k is used for storing the name of the last performed output action $z \in Z_{Ak}$.

(4) Variable ev_k contains the name of the last input event $e \in E_{Ak}$ processed by the automaton A_k, i.e., the name of the event to which A_k reacted.

(5) Variable st_k stores the current state of the automaton A_k. The values of st_k are both basic and intermediate states of the automaton.

(6) Variable *auto* contains the number of the automaton from set A that is active at the current moment, i.e., the number of the last automaton that has received control.

(7) Variable *evnt* stores the name of the input event that came for processing by the nested automaton that has received control. The values of the variable are names of the input events of all nested automata from A and the empty value 0. Variable *evnt* takes value 0 when the input event has been processed or ignored by the automaton (or the automaton in the current state cannot react to this input event). In variable *evnt*, the name of a certain event is written if the control was transferred with this event from the principal automaton to a nested one.

(8) Variable *act* is used for storing the name of the last elementary action that occurred in the automaton system A. The values of variable *act* may be names of input events $a \in E_A$; input queries x and $!x$, where $x \in X_A$; output actions $z \in Z_A$; and special actions 0, *nxty*, and *end*. After an active automaton transits to a new intermediate or basic state, variable *act* contains the label of this intermediate transition. Variable *act* takes value 0 when, being in a basic state, the nested automaton cannot process an input event and ignores it. In this case, the automaton performs the empty action 0 and transits to the same basic state (does not change its state). If *act* = *nxty*, then this means that some nested automaton has transited to the next basic state and transferred control to the principal automaton. Value *end* is used for identification of the deadlock state for the entire automaton system. In order to fulfill the totality condition of the transition relation for the Kripke structure, it is established that there is only one transition from the deadlock state (of the Kripke structure) to the same state. In this case, the action *act* = *end* is performed.

(9) Variables *ev*, *xm*, and *zd* are used for storing names of the last performed input events, input queries, and output actions, respectively, in the framework of the entire system A on the whole.

Then, a state s of the Kripke structure $\boldsymbol{S}_A$ of an automaton model A is given by the following vector of values of variables:

$$(act, auto, evnt, ev, xm, zd, ev_0, xm_0,$$

$$zd_0, y_0, st_0, \ldots, ev_n, xm_n, zd_n, y_n, st_n).$$

In this case, the state s may be viewed as a mapping defined on the set of variables that is used for determining values of these variable in the state s. For example, $s(act)$ is the value of the variable *act* in the state s. At the initial state s_0 of the Kripke structure $\boldsymbol{S}$, all variables y_i and st_i contain the initial states of the corresponding automata A_i, *auto* = 0 (i.e., only the main automaton A_0 is active), *evnt* does not contain events and is set equal 0, and, to all other variables, the initial value *intl* is assigned, which is introduced only for the initialization and is not further used.

The transition relation for the Kripke structure $\boldsymbol{S}_A$ of an automaton model A is defined in accordance with the behavior of automaton system A.

(1) A transition of system A corresponds to only one of the internal transitions of some automaton from A.

(2) When transition conditions are fulfilled, some automaton A_k can perform this transition if it is active in the given state (received control), which is reflected in the value of variable *auto* = k.

(3) For an active nested automaton A_k, the transition with a label e, where e is an input event, may occur if *evnt* = e and the condition for this transition—the predicate over values of variables of states y_i corresponding to this transition is true—is fulfilled. After this transition fires, the values of variables *act*, *evnt*, *ev*, ev_k, and st_k are changed. Variable *evnt* takes zero value, which means that the event has been processed, and, to variables *act*, *ev*, and ev_k, the name of the event e is assigned. The internal state to which the transition was done is placed into st_k. If the main automaton A_0 is active, then the transition with the label e occurs upon fulfillment of the transition conditions, and *evnt* = 0.

(4) If a nested automaton A_k receives control from its principal automaton when an event e stored in the variable *evnt* has occurred but cannot react to it (there does not exist a transition with the label e from the given basic state, or transition conditions for it are not fulfilled), then the empty action (*act* := 0) is performed, the variable *evnt* is set equal zero, and control is transferred to the principal automaton; the number of the principal automaton is placed into the variable *auto*, and all other variables do not change their values.

(5) If a transition with a label x occurs, where x is an input query in the direct or inverse form (x may have the form $!x'$), then the following variable are modified: *act* := x, *xm* := x, $xm_k := x$, and st_k receives the value of the internal state to which the transition has been done.

(6) Upon firing the transition of the automaton A_k with label z, where z is an output action, the assignments *act* := z, *zd* := z, and $zd_k := z$ take place, and st_k receives the value of the internal state to which the transition has been done. Moreover, if z is an output action of the second kind, i.e., $z = A_{k'}(e)$, where $A_{k'}$ is a nested automaton, and e is an input event for the automaton $A_{k'}$ transferred together with the control, then *evnt* receives value e, and the nested automaton $A_{k'}$ becomes active owing to the assignment *auto* := k'.

(7) If a transition with the label *nxty* to a basic state j occurred in the nested automaton $A_{k'}$, then the assignments *act* := *nxty*, $y_{k'} := j$, and $st_{k'} := j$ take place, and the control is transferred to the principal automaton through *auto* := k. If the transition *nxty* occurred in the

main automaton A_0, the variable *auto* = 0 remains unchanged.

(8) If there exists a transition from some state A_k marked by variables z, x, $!x$, or *nxty*, no additional conditions are required for its firing if the automaton is active: *auto* = k.

(9) If the system A of the interacting automata comes to a deadlock state (which means that no transition from the basic state of the main automaton A_0 is possible because of violation of the transition conditions), then a specially introduced transition *act* := *end* occurs, which leads to the same state, and the values of all other variables remain unchanged.

Owing to the Kripke structure constructed, it is possible to use, in specification and verification, predicates over values of the introduced variables for elementary propositions, which makes it possible to express the following properties of the automaton model states.

(1) In each state of an automaton model, there exists a possibility to find out what action was last before the system reached the current state by means of the variable *act*.

(2) It is possible to determine the type of the last action; i.e., it is possible to find out whether an input event, or an input query, or an output action occurred by means of the expressions *act* = *ev*, *act* = *xm*, and *act* = *zd*. For example, if *act* = *zd* in the current state of the automaton model, then the last action before transiting to this state was an output action. To find out what automaton of system A produced this action, expressions of the form $act = zd_i$ are used. Finally, the truth of the expression *act* = z means that the last action upon transiting to the current state was the output action z.

(3) Which automaton is active at the current moment in each state of the automaton model is determined by means of the variable *auto*. For example, if *auto* = 0 in some state of system A, then the main automaton A_0 is active.

(4) In the current state of A, for each automaton A_i, it is possible to determine its last basic state by means of the variable y_i. Moreover, the expression $y_i = st_i$ means that the automaton A_i is in its basic state in the current state of A. Instead of the expression $y_i = st_i$, we will also use the expression $y_i == y_i$ in order to minimize the number of variables used in specification of elementary expressions. Then, whereas $y_0 = 0$ means that the last basic state of the automaton A_0 was state 0, the expression $y_0 == 0$ indicates that the automaton A_0 is in the basis state 0 at the current moment; note that the latter expression is equivalent to $y_0 = 0 \& y_0 = st_0$.

(5) By means of the expression *act* = *end*, it is possible to track deadlock states of the automaton system A, and the transition of one automaton of the system to its new basic state with simultaneous transfer of control to the principal automaton can be tracked by means of the expression *act* = *nxty*.

6. TEMPORAL LOGIC CTL FOR AUTOMATON MODEL

One of the most popular temporal logics for specification and verification of properties of program systems are the CTL (computation tree, or branching time, logic) and LTL (linear-time logic) logics. The use of the LTL for the verification of automaton programs deserves special attention, since any formula in this logic is essentially a Buchi automaton describing (accepting) infinite admissible paths of the Kripke structure, which, in turn, specifies the behavior (all possible implementations) of the automaton model checked for correctness. This allows us to basically use only a simple concept of the "automaton" in the specification and verification of automaton programs. Thus, in a sense, the LTL is a more natural means for specification of properties of automaton programs. On the other hand, the CTL is also widely used in formal verification, and specification in the CTL language is similar in many respects. In what follows, we attempt to estimate convenience of using the CTL (which is considered "less natural" for automaton models) for specifying temporal properties of automaton programs.

CTL formulas for the Kripke structure $\boldsymbol{S}_\mathrm{A}$ of an automaton model A are constructed in accordance with the following grammar:

$$\varphi ::= true|p|\neg\varphi|\varphi \wedge \varphi|\varphi \vee \varphi|$$
$$AX\varphi|EX\varphi|AYi\varphi|EYi\varphi|AF\varphi|$$
$$EF\ \varphi|AG\ \varphi|EG\ \varphi|A(\varphi U\ \varphi)|$$
$$E(\varphi U\varphi)|A(\varphi \tilde{U}\ \varphi)|E(\varphi \tilde{U}\ \varphi),$$

where $p \in P$ is an elementary proposition defined on the set of states of the automaton model A and i in the statements *AYi* and *EYi* is the number of an automaton in A.

The satisfiability relation $\models$ for a state s of the Kripke structure $\boldsymbol{S}_\mathrm{A} = (S, s_0, \longrightarrow, L)$ and a CTL logic formula φ is defined by induction as follows:

- $s \models true$ and $s \not\models false$;
- $s \models p$ for $p \in P \Leftrightarrow p \in L(s)$;
- $s \models \neg\varphi \Leftrightarrow s \not\models \varphi$;
- $s \models \varphi \wedge \psi \Leftrightarrow s \models \varphi$ and $s \models \psi$;
- $s \models \varphi \vee \psi \Leftrightarrow s \models \varphi$ or $s \models \psi$;
- $s \models EX\ \varphi \Leftrightarrow \exists s'\ s \longrightarrow s'$ and $s' \models \varphi$ – there exists a next state of the Kripke structure $\boldsymbol{S}_\mathrm{A}$ in which formula φ is satisfied;
- $s \models AX\ \varphi \Leftrightarrow \forall s'$ from $s \longrightarrow s'$ it follows that $s' \models \varphi$; i.e., in all subsequent states of the structure $\boldsymbol{S}_\mathrm{A}$, the formula φ holds;
- $s \models E(\varphi U\psi) \Leftrightarrow$ for some path π originating from the state $s = \pi(0)$, there exists $j \geq 0$ such that $\pi(j) \models \psi$, and, for all i, $0 \leq i < j$, $\pi(i) \models \varphi$;

• $s \models A(\varphi U \psi) \Leftrightarrow$ for each path π originating from the state $s = \pi(0)$, there exists $j \geq 0$ such that $\pi(j) \models \psi$, and, for all i, $0 \leq i < j$, $\pi(i) \models \varphi$;

• $s \models EF\ \varphi \Leftrightarrow E(true U\ \varphi)$ – there exists a path from s in the Kripke structure $\boldsymbol{S}_A$ that passes through a state in which φ holds;

• $s \models AF\ \varphi \Leftrightarrow A(true U\ \varphi)$ – any path from the state s in the Kripke structure $\boldsymbol{S}_A$ necessarily passes through a state in which φ holds;

• $s \models E(\varphi \tilde{U}\ \psi) \Leftrightarrow$ for some path π originating from the state $s = \pi(0)$ and for any $j \geq 1$ such that $\pi(j) \not\models \psi$, there exists i, $0 \leq i < j$, such that $\pi(i) \models \varphi$ (for some path, formula ψ is true and must remain true until φ is true);

• $s \models A(\varphi \tilde{U}\ \psi) \Leftrightarrow$ for each path π originating from the state $s = \pi(0)$ and for any $j \geq 1$ such that $\pi(j) \not\models \psi$, there exists i, $0 \leq i < j$, such that $\pi(i) \models \varphi$ (for each path, formula ψ is true and must remain true until φ is true);

• $s \models EG\ \varphi \Leftrightarrow E(false \tilde{U}\ \varphi)$ – there exists a path from the state s in the Kripke structure $\boldsymbol{S}_A$ through which φ holds (in each state of the path);

• $s \models AG\ \varphi \Leftrightarrow A(false \tilde{U}\ \varphi)$ – through any path (in each state of the path) from the state s in the Kripke structure $\boldsymbol{S}_A$, φ holds;

• $s \models EYi\ \varphi \Leftrightarrow EX\ E(y_i! = y_i U y_i == y_i \& \varphi)$ – among the basic states of an automaton A_i of system A that immediately follow state s, there exists a state such that formula φ holds;

• $s \models AYi\ \varphi \Leftrightarrow AX\ A(y_i! = y_i U y_i == y_i \& \varphi)$ – for each basic state of an automaton A_i of system A that immediately follow state s, the formula φ must hold.

The last two operators are introduced in order that to be able to express properties related mostly to the states of automata A_i from A in a most natural way.

In addition to logical connectives $\wedge$ and $\vee$, the connectives $\longrightarrow$ and $\longleftrightarrow$ are traditionally used:

• $\varphi \longrightarrow \psi \equiv \neg\varphi \vee \psi$;

• $\varphi \longleftrightarrow \psi \equiv (\varphi \longrightarrow \psi) \wedge (\psi \longrightarrow \varphi) \equiv (\neg\varphi \vee \psi) \wedge (\neg\varphi \vee \varphi)$.

For temporal operators, the following relations hold (they follow immediately from the definitions):

• $AX\ \varphi \equiv \neg EX \neg\varphi$,

• $AF\ \varphi \equiv \neg EG \neg\varphi$,

• $AG\ \varphi \equiv \neg EF \neg\varphi$,

• $A(\varphi \tilde{U}\ \psi) \equiv \neg E(\neg\varphi U \neg\psi)$,

• $E(\varphi \tilde{U}\ \psi) \equiv \neg A(\neg\varphi U \neg\psi)$.

By using the last two relations, it is possible to get rid of the operators $E\tilde{U}$ and $A\tilde{U}$ in temporal formulas, since their definitions are rather involved and may result in difficulties in verbal interpretation (understanding) of formulas.

In what follows, along with the symbols $\wedge$ and $\vee$, we will also use the symbols & and |.

A temporal property given by a temporal CTL formula φ is considered true for the Kripke structure $\boldsymbol{S}_A = (S, s_0, \longrightarrow, L)$ of an automaton model A if and only if $s_0 \models \varphi$ holds.

7. TEMPORAL PROPERTIES OF AUTOMATON MODELS

We consider several examples of temporal properties that are common for all particular hierarchical automaton models (for any hierarchical system of interacting automata).

"Deadlock state." The property that describes possibility of coming to a deadlock state is applicable to any automaton program (to a model of any automaton program of the considered type). In the framework of the above-described Kripke structure, this property can be described in the language of the temporal CTL logic as follows:

$$EF\ act = end.$$

This formula means that there exists a path from the initial state s_0 of the Kripke structure to the state from which the transition with the label *end* took place. By construction of the Kripke structure $\boldsymbol{S}_A$, such a state is a deadlock, since the transition *end* is added when no other transitions from this state exist. In view of a hierarchical structure of the automaton models, a deadlock situation occurs only when it is impossible to leave the basic state of the main automaton A_0 because of violation of the transition conditions or if there are no transitions from this state at all.

"Deadlock state of a nested automaton." In the previous example, the deadlock state of the whole system was due to entering of the automaton A_0 into its basic state from which there is no way out. In this example, we consider the property that expresses availability of a deadlock in the nested automata. Suppose that the Kripke structure $\boldsymbol{S}_A$ transited to a new state by means of a transition of a certain automaton A_k from A to its next basic state with simultaneous transfer of control to the principal automaton. Next, suppose that, for any paths from this state of the Kripke structure, any input event passed to the automaton A_k together with the control is ignored either because of violations of the transition conditions or because of lack of transitions marked by this input event. Hence, the nested automaton ran into some basic state and will never leave it. This basic state at the given time moment is said to be a deadlock state

for the automaton A_k, although this may be not true in a different situation. The property of being deadlock for a basic state of the automaton A_k is specified as follows:

$$EF\,AG(auto = k \longrightarrow act! = ev_k) \text{ or}$$

$$EF\,AG(auto = k\&evnt! = 0 \longrightarrow AX\,act = 0) \text{ or}$$

$$EF\,AG(auto = k\&y_k == y_k \longrightarrow AX\,act = 0).$$

"Lost automaton." Consider one more kind of the deadlock state of a nested automaton. Starting from some moment (from some state of a path in a Kripke structure $\boldsymbol{S}_A$), a nested automaton A_k will never receive control. Thus, starting from some moment, the automaton A_k will remain in one of its basic states for ever. This property is expressed as

$$EF\,AG\,auto! = k.$$

Below, we consider some examples of temporal properties related to a particular automaton model of a coffee maker control system (Section 3).

8. SPECIFICATION OF TEMPORAL PROPERTIES OF AUTOMATON MODEL OF COFFEE MAKER CONTROL SYSTEM

We demonstrate convenience and usefulness of using the introduced Kripke structure and temporal CTL logic for specification and verification of properties of automaton model of a coffee maker control system.

Let us consider the following property of the coffee maker control system formulated in a natural language.

"Correct interrupt of coffee maker operation by user." If the valves and the heater are not broken, and the coffee maker boils coffee in an ordinary mode, then, after pressing the "C" button, the main automaton A_0 of the coffee maker control transits to the state "Interrupted by user," and all nested automata must return to their initial states before the coffee maker starts to work again.

Let us rephrase this property using elements of the automaton model.

If, for $y_{31}! = 4$, $y_{32}! = 4$, $y_2! = 4$, and $y_0 == 2$, event e_{02} occurs, then the automaton A_0 will necessarily process it (will not ignore) and transit to the state $y_0 = 4$. When event e_{01} occurs for $y_0 = 0$, the system will be in the state $y_{31} = 0$, $y_{32} = 0$, $y_2 = 0$, and $y_1 = 0$.

Let us consider the Kripke structure generated by the analyzed system of interacting automata and rewrite the property using the concept of path in the Kripke structure of the automaton model.

For all paths in the model, we have the following. If, in a state of a path, the expression $y_{31}! = 4\&y_{32}! = 4\&y_2! = 4\&y_0 == 2$ is true and event e_{02} occurs at the next moment of time, then this event will necessarily be processed, and the next basic state of the automaton A_0 will be $y_0 == 4$. When the automaton A_0 comes to the state $y_0 == 0$ sometime later and process event e_{01}, this will imply that the system of automata has already been in the state $y_{31} == 0\&y_{32} == 0\&y_2 == 0\&y_1 == 0\&y_0 == 0$ before processing e_{01}.

Let us replace all propositions over paths by temporal operators. We will obtain the following temporal CTL formula for the described property:

$$AG(y_{31}! = 4\&y_{32}! = 4\&y_2! = 4\&y_0 == 2$$

$$\longrightarrow EX\,act = e_{02}$$

$$\&AX\,(act = e_{02}$$

$$\longrightarrow AY_0\,(y_0 == 4$$

$$\&AG\,(y_0 = 0\&act = e_{01}$$

$$\longrightarrow y_{31} == 0\&y_{32} == 0\&y_2 == 0\&y_1 == 0)))).$$

In a similar way, we consider and specify a number of other properties of the coffee maker control system.

"Fault determination." If the heater or one of the valves is broken, the coffee maker (main automaton A_0) will necessarily transit to the state "Fault."

(1) For each path (or implementation) of the Kripke structure of the considered system of interacting automata, the following requirement must be satisfied. If the systems turns to the state $(y_{31} = 4|y_{32} = 4|y_2 = 4)\&y_0 = 2$, then the next state of the automaton A_0 that is different from the state $y_0 = 2$ ("Boils coffee") will necessarily be the state $y_0 = 5$.

(2) $AG((y_{31} = 4|y_{32} = 4|y_2 = 4)\&y_0 = 2 \longrightarrow A(y_0 = 2Uy_0 = 5))$.

"Fault indication." If the main automaton A_0 turned to the state "Fault," then a message on the fault type is shown on the display (the situation where a fault takes place but the user is not informed on its type is not permitted).

(1) It is advisable to rewrite this property in a negative form as follows. There exists a path in the Kripke structure that leads to the state $y_0 = 5$, and, through this path, no messages on the valve or heater damage are displayed (none of the actions $act = z_{35}$ or $act = z_{25}$ takes place).

(2) $!E((act! = z_{35}\&act! = z_{25})Uy_0 = 5)$.

"Fault reset." If the automaton A_0 is in the state "Fault," then pressing the "C" button resets all automata, including A_0, into their initial states before the coffee maker starts to work again.

(1) Through all paths of the Kripke structure, if $y_0 = 5\&act = e_{02}$ in a state of a path, then, in any case, the next state of A_0 will be $y_0 = 0$, and the event e_{01} for $y_0 = 0$ may be processed when $y_1 = 0\&y_2 = 0\&y_{31} = 0\&y_{32} = 0$ holds.

(2) $AG(y_0 = 5\&act = e_{02} \longrightarrow AY_0(y_0 = 0\&AX(act = e_{01} \longrightarrow y_1 = 0\&y_2 = 0\&y_{31} = 0\&y_{32} = 0)))$.

"Display reset after fault." If the automaton A_0 is in the state "Ready for operation" after a fault, then no fault messages are shown on the coffee maker display (including the boiler, heater, and valve displays).

(1) If a fault message was shown on the coffee maker display (including the boiler, heater, and valve displays), then, before A_0 turns to the initial state $y_0 = 0$, this message should be deleted from the display. In this case, it is advisable to write this property in a negative form, expecting that this condition is violated on some path of the Kripke structure.

(2) $!EF((act = z_{35}\&E(act! = z_{36}Uy_0 = 0))|$

$(act = z_{25}\&E(act! = z_{26}Uy_0 = 0))|$

$(act = z_{08}\&E(act! = z_{06}Uy_0 = 0))).$

"Availability of water." The coffee maker will never boil coffee without water (accordingly, the boiler control automaton A_1 will never turn from the state "Ready for boiling the next portion" to the state "Boiling" if the amount of water is not sufficient).

(1) Let us rewrite the proposition in a negative form. There exists a path in the Kripke structure from the state $y_1 == 2$ to the state $y_1 = 4$ through which the water availability sensor has not been polled or the result of the poll was negative.

(2) $!EF(y_1 == 2\&E(act! = x_{13}Uy_1 = 4))$.

"Inadmissibility of the heater overheat." When the heater reaches the maximum temperature, it always turns off: there does not exist a situation (endless process) when the heater of the coffee maker will heat endlessly after exceeding the temperature threshold.

(1) If the coffee maker runs, no overheat may take place before it returns to the initial state. In other words, during the coffee maker operation, there may be no situation where the automaton A_2, being in the state $y_2 == 2$ ("Heater is turned on"), will have no chance to check its transition conditions for an infinite long time (will not be able to send a query to the temperature sensor or will not receive the command to turn off). Such a situation is possible in two cases. In the first case, the main automaton A_0 falls into a state from which it can escape only if the button is pressed or if it addresses (by the system timer) the control object with a query of parameters, whereas the automaton A_2 is in the state $y_2 == 2$. Then, the automaton A_2 can potentially endlessly wait for signals e_0 and e_{22} from the automaton A_0, which will result in the heater overheat. In the second case, there exists an endless path in the Kripke structure of the automaton model on which the automaton A_2 in the state $y_2 == 2$ also does not receive events e_0 and e_{22} from the main automaton A_0. This case includes the possibility that the automaton system falls into a deadlock state under the condition $y_2 == 2$, since the endless path leading to the deadlock state, which further passes through it because of the loop $act = end$, is taken into account. It is important to note that the automaton A_0 does not contain any queries x.

(2) $!EF(y_2 == 2\&auto = 0\&y_0 == y_0\&AX(act! = e_0\&act! = end))\&!EF\ EG(y_2 == 2\&(auto = 2 \longrightarrow AX(act! = e_0\&act! = e_{22})))$.

"Initial state." If the automaton A_0 turned to the initial state "Ready for operation," then all nested automata are already in their initial states.

(1) For all paths of the Kripke structure of the automaton model, if the automaton A_0 came to the state $y_0 == 0$, then all nested automata are in the initial states: $y_1 == \&y_2 == 0\&y_{31} == 0\&y_{32} == 0$.

(2) $AG(y_0 == 0 \longrightarrow y_1 == 0\&y_2 == 0\&y_{31} == 0\&y_{32} == 0)$.

"Reactivity condition." The coffee maker control system will never come to a state in which it does not react to events of the system timer or to pressing buttons "Ok" and "C."

(1) There does not exist a path to a state that the system cannot leave. In other words, the system never comes to a deadlock. Since the Kripke structure of the automaton model possesses the total transition relation, each deadlock state has only one transition, the transition to itself. In this case, the action $act = end$ is performed.

(2) $!(EF\ act = end)$.

Thus, the above examples show that intelligible system properties, which have rather involved formulation in a natural language, can successfully be specified as CTL formulas, which makes it possible to automatically verify them for the model under consideration. Note that similar properties written in the LTL logic language look simpler.

9. MODEL REDUCTION

Since automated verification by the model checking method suggests using search methods, it is important to generate the Kripke structure of the automaton model that contains as few states as possible but still allows us to verify temporal properties of the automaton model.

In the given case, it is possible to reduce the introduced Kripke structure of the automaton model with respect to the verified temporal properties given by CTL formulas.

Let us describe several examples when the reduction of the model with respect to a formula is advisable.

If elementary propositions in a temporal formula are predicates defined only with the use of expressions over basic states of the form $y_i == k$, then we may not consider intermediate states of the model that have nothing to do with output actions of the second kind (Section 2): all intermediate states not related to control transfer between the automata can be removed from the model.

For example, consider the formula

$$AG(y_0 == 0 \longrightarrow y_1 == 0\&y_2 == 0$$

$$\&y_{31} == 0\&y_{32} == 0).$$

If this formula is true (or false) for some Kripke structure, then it remains true (or false) for the reduced Kripke structure obtained from the original one by

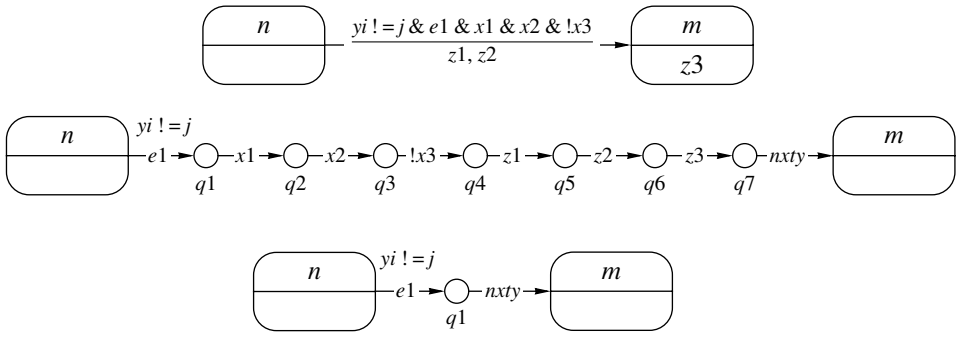

Fig. 8. Example of reduction of the number of intermediate states in the Kripke structure for automaton transitions not containing output actions of the second kind.

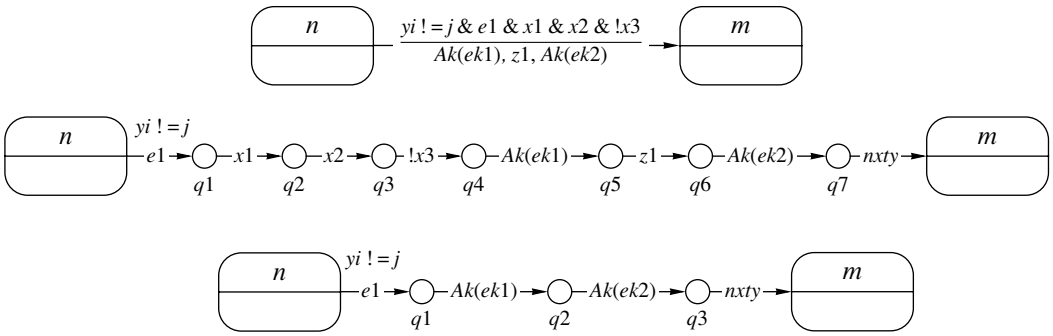

Fig. 9. Example of reduction of the number of intermediate states in the Kripke structure for automaton transitions containing output actions of the second kind.

means of the transformation (reduction) shown in Fig. 8. In this figure, the original automaton transition does not contain output actions of the second kind, which made it possible to reduce the number of the intermediate states to one for a transition between the basic states of the automaton.

Figure 9 shows an example of reduction of the Kripke structure in the case where the control is transferred to a nested automaton during the automaton transition.

Consider one more example of the formula for which reduction of the dimension of the Kripke structure of the automaton model is possible:

$$EF(act = z_2 \& EX\, E(act! = z_2 U y_0 = 0)).$$

This formula expresses the following property. There exists a path in the Kripke structure containing a transition marked by the output action z_2, and, after this transition, through the remaining part of the path to the state $y_0 = 0$, the transition with the label z_2 is not met anymore. For this property, transitions with labels different from z_2 are not of interest and can be depersonalized, and a sequence of several impersonal transitions going one after another can be replaced by one transition or be even excluded from the Kripke structure. Figure 10 shows an example of reduction of the Kripke structure by the above formula for the automaton transition that has no output actions of the second kind.

Thus, after analysis of a temporal formula, it is possible to construct the Kripke structure of the automaton model with fewer number of states that is invariant with respect to the formula, which makes it possible to verify the properties of the automaton model given by this formula in less time.

In connection with this, of interest are classes of temporal properties for which reduction of the Kripke structure of the automaton model is most efficient.

10. PRACTICAL IMPLEMENTATION

The definition of the Kripke structure of the automaton model and the property specification described make it possible to apply the model checking method for verification of automaton programs. For this purpose, it makes sense to use already existing application packages for verification developed and supported by the leading scientific laboratories for a long period (more than ten years).

However, the problem is that each verificator has its own formalism for specifying models and its own method of generation of the Kripke structure for this model. Moreover, the verificators have different modifications (implementations) of temporal logic, which may be less expressive than the temporal CTL logic considered above.

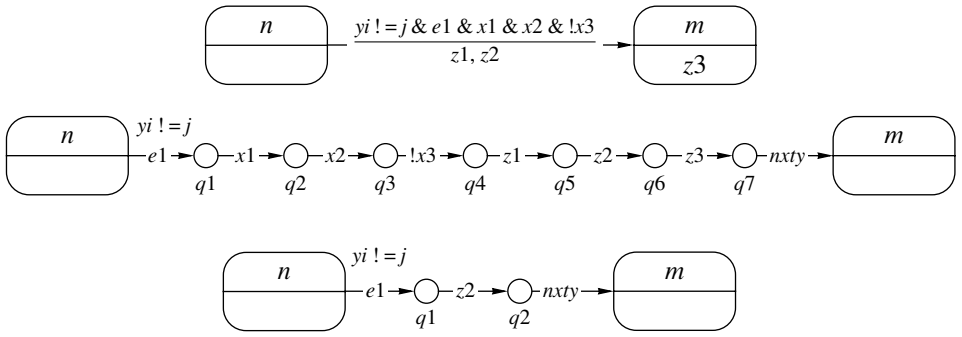

Fig. 10. Reduction of the number of intermediate states in the Kripke structure with respect to formula $EF(act = z_2 \& EX\ E(act! = z_2 U y_0 = 0))$ for automaton transitions not containing output actions of the second kind.

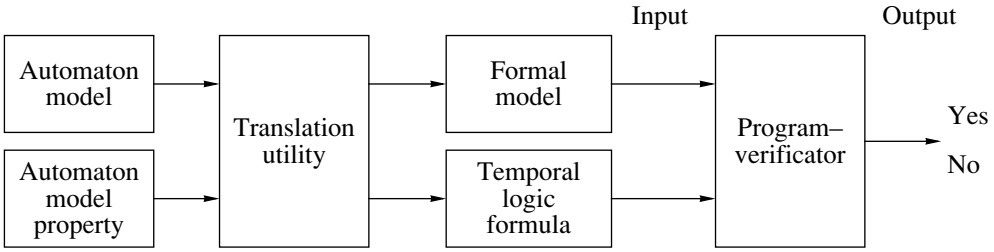

Fig. 11. Use of the model checking method for verification of automaton programs.

One faces a difficult problem of adequate specification of the Kripke structure using the means of the existing verificators. Speaking of adequacy, we mean generation of the Kripke structure for the given formal model that possesses all properties of the original automaton model and does not possess any other properties that are not inherent in the original model. It should be guaranteed that, after checking properties for the model specified in the framework of the program–verificator, the result of this checking will uniquely be applicable to the properties of the original automaton model. Moreover, when choosing the verificator, one should take into account expressiveness of the temporal logic implemented in it in order to be able to express types of the automaton model properties described above.

Thus, after selection of an adequate and expressive verificator, practical implementation of automated checking properties of automaton programs can be represented in the form shown in Fig. 11.

An important feature of this scheme is the utility of adequate translation of the automaton model and its properties into such a formal model and specification that are allowed by the interface of the program–verificator. In accordance with the requirements, the translation utility must always ensure correct and unambiguous correspondence between results of checking properties by the verificator and the truth of this property of the automaton model.

11. CONCLUSIONS

An automaton program is a very convenient object for verification by the model checking method. Properties of the automaton program system are formulated and specified in a natural and clear way. They are easily related to the interacting automata that specify logic of the automaton program, since elements of the control automata are either explicitly expressed states of the control object or comprehensible actions on it. The verification of properties is carried out in the terms that naturally follow from the automaton model of the program. The elementary propositions in the context of these properties are defined on the model elements, i.e., on events, input and output actions, and states. If testing reveals an error after the verification by the model checking method, then it will, most likely, refer to incorrect program implementation of the output actions rather than to the violation of the program logic, which will not require global reengineering of the automaton program (the correction will reduce to local modifications of one or several, usually small, procedures, the correctness of which can then be proved by the deductive analysis method).

The development of formal methods and technologies for simulation, specification, and verification of automaton programs and construction of an integrated program complex on this basis will allow us to design and implement reliable program systems for controlling crucial objects. For this purpose, it makes sense to use already existing packages of applied program–ver-

ificators developed and supported by leading foreign and Russian scientific laboratories and centers. The majority of such verification tools are provided free for researchers. However, there exist commercial versions of program–verificators that can be used for industrial purposes by themselves and in combination with other tools for program analysis and verification. Research in this direction [13–15] resulted in the creation of a verification program prototype called "Automaton program simulation and verification system" (Vinogradov, R.A., Kuzmin, E.V., and Sokolov, V.A., certificate no. 2007611856).

It is important to note that, in construction of reactive real-time systems on the basis of the automaton approach to programming, the classical model checking method is not sufficient for analysis of complex temporal properties (properties with time constraints). In this case, it is required to apply the existing methods and tools for the verification of real-time systems, which are extensions of the model checking method. In the construction and modeling of temporal systems, the main focus is placed on the time interpretation. In the automaton programming of synchronous real-time systems, it makes sense to construct and verify models with discrete time. In this case, the specification is implemented in a language of real-time temporal logic. The temporal logic extended with regard to the discrete real time makes it possible to express such properties as "it is always true that q follows p not later than in three units of time." An example of such a temporal logic is the RTCTL (real-time CTL), which is used for specification of properties in the VERUS verificator. On the other hand, the continuous time is natural for asynchronous systems, since the time interval separating events may be as small as desired. Standard formalism for modeling and analysis of asynchronous real-time systems is provided by timed automata.

Verification of automaton programs with the use of models based on timed automata can be done, for example, by means the UPPAAL [16] and KHRONOS [17] tools.

Finally, for modeling of automaton programs, linear hybrid automata (generalization of timed automata) can be used. In this case, only partial algorithms are used for solving the verification problems. However, as noted in the manual to the HyTech system [18] designed for verification of linear hybrid automata, partial algorithms used in HyTech converged in the majority of cases. Moreover, there exists a wide class of hybrid systems for which iteration methods and analysis procedures are decidable (always converge). Thus, the linear hybrid automata present a borderline (in terms of expressiveness) formalism, which is still applicable to modeling and analysis of the automaton programs.

ACKNOWLEDGMENTS

We are grateful to A.A. Shalyto for attracting our attention to the problematics of the automaton programming discussed in this paper.

REFERENCES

1. Shalyto, A.A., SWITCH-Technology, *Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (Algorithmization and Programming of Logic Control Problems), St. Petersburg: Nauka, 1998, http://is.ifmo.ru/books/switch/1/
2. Shalyto, A.A., Automaton Program Design. Algorithmization and Programming of Logic Control Problems, *Izv. Ross. Akad. Nauk, Teor. Sist. Upr.*, 2000, no. 6, pp. 63–81, http://is.ifmo.ru
3. Shalyto, A.A., Algorithmization and Programming for Logic Control Problems and "Reactive" Systems, *Avtom. Telemekh.*, 2001, no. 1, pp. 3–39, http://is.ifmo.ru
4. Shalyto, A.A. and Tukkel, N.I., Programming with Explicit Selection of States, *Mir PK*, 2001, no. 8, pp. 116–121, no. 9, pp. 132–138, http://is.ifmo.ru
5. Shalyto. A.A. and Tukkel, N.I., SWITCH-Technology: An Automated Approach to Developing Software for Reactive Systems, *Programmirovanie*, 2001, no. 5, pp. 45–62. [*Programming Comput. Software* (Engl. Transl.), 2001, vol. 27, no. 5, pp. 260–276].
6. Gries, D., *The Science of Programming*, New York: Springer, 1981. Translated under the title *Nauka programmirovaniya*, Moscow: Mir, 1984.
7. Clarke, E. M., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, 1999. Translated under the title *Verifikatsiya modelei program: Model Checking*, Moscow: MTsNMO, 2002.
8. CPNTools, http://www/daimi.au.dk/CPNTools/
9. SPIN, http://spinroot.com/spin/whatispin.html
10. *SMV, Symbolic Model Verificator*, Carnegie Mellon Univ., http://www.cs.cmu.edu/-modelchek/smv.html
11. *CADP, Construction and Analysis of Distributed Process*, http://www.inrialpes.fr/vasy/cadp/
12. Kessel', S.V., *Razrabotka sistemy upravleniya kofevarkoi na osnove avtomatnogo podkhoda*, (Development of Coffee Machine Control System on the Basis of Automaton Approach), SPbGU ITMO, 2003, http://is.ifmo.ru/projects/coffee2/
13. Vinogradov, R.A., Kuzmin, E.V., and Sokolov, V.A., Verification of Automaton Programs by CPN/Tools, *Modelirovanie i analiz informatsionnykh sistem*, Yaroslavl': YarGU, 2006, vol. 13, no. 2, pp. 4–15.
14. Kuzmin, E.V. and Sokolov, V.A., Some Approaches to Verification of Automaton Programs, *Sbornik dokladov seminara Go4IT—shag k novym tekhnologiyam Interneta* (A Collection of Seminar Reports Go4IT—A Step to New Internet Technologies), Moscow: Institut Sistemnogo Programmirovaniya RAN, 2007, pp. 43–48.
15. Kuzmin, E.V. and Vasil'eva, K.A., Verification of Automaton Programs Using LTL, *Modelirovanie i analiz informatsionnykh sistem*, Yaroclavl': YarGU, 2007, vol. 14, no. 1, pp. 3–14.
16. UPPAAL, http://www.uppaal.com
17. KRONOS. http://www-verimag.imag.fr/TEMPORISE /kronos
18. HyTech, http://embedded.eecs.berkeley.edu/research/hytech/