Computer Science Journal of Moldova, vol.4, no.1(10), 1996

Parallel logical control algorithms: verification and hardware implementation

A.Zakrevskij

Abstract

A formal language (PRALU) has been proposed for representation of parallel algorithms for logical control. The paper contains a short description of its syntax and semantics, methods of checking PRALU-algorithms for correctness and methods for their hardware implementation. They include using suggested parallel automata as a standard form of algorithms, coding their partial states by ternary vectors, and obtaining appropriate minimized systems of logical equations of the sequent type. The latter ones could be easily implemented by logic nets with matrix structure.

1 Introduction

A proper interaction between components of computers, industrial systems, communication nets, robotic complexes, etc., can be provided by logical control devices. They are discrete dynamic systems, exchanging information with controlled objects by means of logical variables. Generally speaking, these devices can be looked at as asynchronous systems controlling parallel and concurrent processes. They are diverse and their design implies the solution of numerous hard combinatorial problems.

It is natural to begin the design of these devices with formulating a logical control algorithm, deriving it from a notion of the behaviour of a system that has to be put under control. The formal language PRALU has been proposed for the description of such algorithms [1,2].

Any logical control algorithm is represented in PRALU as a set of chains — linear algorithms composed of waiting and acting operations. These chains can be fulfilled in parallel and interact both directly, with the help of a mechanism analogous to the Petri net [3], and informationally, by means of some common Boolean variables. The representation of hierarchical structures is possible.

The elaboration of logical control algorithms entails their verification. The verification is decomposed into the syntax checking and the testing for correctness carried out automatically, and also semantic testing in interactive mode when the algorithm is simulated on a computer. The check-up of correctness is reduced to a great extent to that of safety and liveliness of ordinary Petri nets, which turn out to be extended nets of free choice [4] — under the restrictions adopted in PRALU.

Rather complicated problems arise when synthesising switching circuits implementing parallel logical control algorithms. Their solution is facilitated by preliminary transformation of control algorithms to "automata" representation using traditional notions of state and transition. But the classical finite automaton turns out to be inadequate, and that is why the notion of the parallel automaton (a peculiar hybrid of the sequent expressions with the Petri net) has been introduced. A parallel automaton can find itself simultaneously in several states (partial), and the transitions are defined not between single states but between some groups of states. Such a model is more compact in practical situations compared with the classical one and principally cannot be reduced to the latter in case of the asynchronous control.

The well-known problem of state assignment is much more difficult in the case of parallel automata. It has been reduced to coding of partial states by ternary vectors of minimum length, in such a way that parallel states (that can exist simultaneously) have to be represented by compatible (not orthogonal) vectors.

Having coded the states of a parallel automaton we get a system of sequents $k_i \vdash k_i'$ with elementary conjunctions k_i and k_i'. Each sequent is interpreted as follows: if at some arbitrary moment $k_i = 1$, then immediately after that $k_i' = 1$ has to be satisfied.

Going further, we have to implement this system of sequents, synthesising a corresponding switching circuit in some basis, for instance in PLAs, that are the most convenient for that purpose.

2 PRALU language

PRALU language has been proposed for the description of asynchronous parallel logical control algorithms in terms of input and output Boolean variables of control devices.

Two operations, action and waiting, defined on an elementary conjunction k, are the main elements of this language. The action operation $\to k$ assignes to the variables of k such values that satisfy the equation $k = 1$. The waiting operation $-k$ does not change any values but waits until k becomes equal to 1, and only then its fulfillment terminates.

The sequences of action and waiting operations are considered as linear algorithms. In general case, a logical control algorithm is presented as an unordered set of expressions α_i, or chains, of the form

$$\mu_i : -k_i \quad l_i \to \upsilon_i,$$

where l_i denotes some linear algorithm (in particular, it may consist of only one action operation or be "empty"), and μ_i and υ_i respectively denote the initial and terminal chain labels represented by some subsets of the set $M = \{1, 2, \ldots, m\}$.

The chains are controlled by means of the variable starting set N which takes as its current value N_t some subset of the set M. If the condition $(\mu_i \subseteq N_t)\&(k_i = 1)$ is satisfied at some instant t, the chain α_i will be started. In that case μ_i is expelled from N_t, then the linear algorithm l_i is executed and, in conclusion, the transition operation $\to \upsilon_i$ will be performed adding the set υ_i to the set N. As a result, the new value of N becomes equal to $(N_t \backslash \mu_i) \cup \upsilon_i$. The initial value N_0, which is one-element as a rule (for example, $N_0 = \{1\}$), is assigned to the set N when starting the algorithm. Before the algorithm is started, all chains are assumed to be passive (i.e. they are not being performed).

This mechanism is strong enough for alternative branching and for making the processes concurrent, for the convergence of alternative branches and for the merging of concurrent ones. The chains which can be performed simultaneously are called parallel chains; the algorithm with such chains is also called a parallel one.

Alternative branching is ensured by the following constraint introduced into the language:

$$(i \neq j)\&(\mu_i \cap \mu_j \neq \emptyset) \to (k_i \& k_j = 0).$$

For convenience, chains with similar initial labels are united in a sentence wherein the chain-alternatives are written one under another.

The other constraint

$$(i \neq j)\&(\mu_i \cap \mu_j \neq \emptyset) \to (\mu_i = \mu_j) \tag{1}$$

simplifying the interpretation of algorithms follows from the graphical definitions, which facilitate application of PRALU language in engineering practice.

It is desirable to single out for further consideration the two-terminal algorithms (with initial and final values of the starting set), and also cyclic algorithms to which two-terminals are easy to reduce. The two-terminals may be used as blocks in hierarchical algorithms (instead of some action operations); the cyclic algorithms are widely used when representing production processes and are of convenience in the investigation of the algorithm correctness problem.

We shall illustrate the PRALU language with an example of a cyclic algorithm

$$\begin{aligned}
1 &: -u \to ab - \bar{u} \to 2.3\\
2 &: -\bar{v}w \to \bar{b}c - \bar{w} \to b \to \bar{c} \to 2\\
&\quad -v \to \bar{a}c \to 4.5\\
3 &: -uv \to d \to 6\\
4 &: -\bar{u}\bar{v} \to a - u \to \bar{a} \to 4\\
&\quad -u \to \bar{a}b \to 7\\
5 &: -\bar{v}w \to \bar{c} \to 8\\
6.7.8 &: \to \bar{a}\bar{d} - \bar{w} \to 1
\end{aligned}$$

The algorithm is initiated by the assignment of the value $\{1\}$ to the starting set N. First the chain 1 is executed: we wait until the input Boolean variable u takes the value 1, then give the same value to the output variables a and b, then wait for the event $u = 0$, and after that split the process starting both the sentences 2 and 3 simultaneously. The sentence 2 contains two chains, beginning with the waiting operations $-\bar{v}w$ and $-v$. Only one of those chains is initiated — immediately after the condition $\bar{v}w = 1$ or $v = 1$ is satisfied. The other sentences are executed in the same manner when started. At last the chain with the initial label 6.7.8 is executed — after the fulfillment of the chains with the final labels 6,7 and 8. After that the process is repeated.

3 Parallel automaton

When solving a number of problems pertaining to the analysis of hierarchical algorithms and the synthesis of some structures realizing them, it is expedient to reduce the algorithm (by cutting "long" chains) to a standard form in which all chains look similar:

$$\mu_i : -k'_i \to k''_i \to \nu_i.$$

Presented in such a form, PRALU-algorithm may be considered as an automaton. It is not however the traditional sequential automaton, but the parallel automaton with partial states (these are the elements from M) that can find itself in several of these states at the same time [5].

The chains are interpreted as follows: if the automaton is in the states listed in the label μ_i (and, perhaps, in some others) and if the variables in the conjunction term k_i' assume the values at which $k_i' = 1$, then the variables from the term k_i'' accept the values at which $k_i'' = 1$, and the states forming the label μ_i are substituted by the states from the label ν_i. Thus a transition is made between groups of states rather than between separate states. The states not listed in μ_i and ν_i will not be affected by any transitions: if the automaton is in any of them, it will remain in it, otherwise it will not enter this state.

This model differs basically from the classical sequential automaton model. In the case of synchronous interpretation, it may be reduced to the latter, yet practically this is not desirable: for example, a parallel automaton with $3n$ partial states may generate a sequential automaton with 3^n states (when the transition graph of the parallel automaton consists of n parallel chains, having 3 partial states each). In the case of asynchronous interpretation, reflecting the local interactions between some variables (these interactions can take place when the transients corresponding to the change of values of other variables have not yet been attenuated) such reduction is not possible at all [5].

Two mechanisms of interaction between chains are used in the model.

One of them is a simplified version of the ordinary Petri net [3] and is called α-net. It is defined as a system consisting of the set N_0 and the unordered set of pairs $\mu_i \to \nu_i (i = 1 \ldots n; N_0, \mu_i, \nu_i \subseteq M)$, obeying the constraint (1). By analogy with the Petri net it is interpreted as a dynamic model with the set of places M, the initial state N_0 and a current state N_t that is changeable on transitions $\mu_i \to \nu_i$ (denoted hereinafter as τ_i). It is supposed that transitions τ_i may occur, one by one, when the conditions $\mu_i \subseteq N_t$ are satisfied, and that execution of τ_i consists in replacing the current value N_t by $(N_t \backslash \mu_i) \cup \upsilon_i$.

Let us call an α-net safe if for any reachable (from N_0) state N_t and for any transition τ_i the condition

$$(\mu_i \subseteq N_t) \to ((N_t \backslash \mu_i) \cap \nu_i = \emptyset)$$

is satisfied.

The following theorem establishes a relationship between α-nets and expanded nets of free choice (EFC-nets) investigated by Hack [4].

Theorem 1 *Safe α-nets are equivalent to safe expanded nets of free choice.*

The second mechanism of interaction between chains of a parallel automaton is presented by pairs of operations $-k_i' \to k_i''$. Such a pair is similar to the sequent $k_i' \vdash k_i''$, specifying the condition-event relationship between simple events represented by the conjunction terms k_i' and k_i'': the event k_i' gives rise to the event k_i''. A system of such pairs may be interpreted as a simple sequent automaton [6]. Obviously, the chain α_i is able to control the chain α_j if $\sigma_i'' \cap \sigma_j' \neq \emptyset$, where σ_i'' and σ_j' denote the sets of Boolean variables in k_i'' and k_j', respectively. If $\sigma_i'' \cap \sigma_j' \neq \emptyset$ and $\sigma_j'' \cap \sigma_i' \neq \emptyset$, then the chains α_i and α_j will be able to carry on a dialog.

Therefore, the parallel automaton is a peculiar combination of two formal tools: the α-net and the simple sequent automaton. The second one is more powerful, and the information contained in an α-net can easily be "pumped" into a system of simple sequents. However, this operation is essentially complicated when the system has to be minimized.

4 Correctness verification

Of great importance is the verification of algorithm correctness, i.e. its quality which may be established formally, without knowing the specific purpose of the algorithm and just on the basis of general requirements to the algorithms of the class under consideration.

Correctness of parallel logical control algorithms was defined in [7] as the combination of five qualities: consistency, persistency, irredundancy, recoverability and self-coordination.

The algorithm is consistent if any of its parallel chains, α_i and α_j, are compatible. In particular, in that case the condition $k_i'' \wedge k_j'' \neq 0$ must be satisfied. The algorithm is persistent if the completion of one of the parallel chains being performed does not destroy conditions for executing the others ($k'_i \wedge k''_j \neq 0$ and $k''_i \wedge k'_j \neq 0$ for parallel chains). The algorithm is irredundant if it contains no chains that can never be executed. The algorithm is recoverable if it can return to its initial state from any reachable state. This requirement is characteristic of cyclic algorithms and is identical to reenterability in the theory of programming. The algorithm is self-coordinated if none of its chains can be reinitiated during its execution (for example, a new workpiece cannot be fed to the machine-tool until the previous one has been machined).

Note that irredundancy and recoverability requirements are applicable both to parallel and purely sequential algorithms, whilst consistency, persistency and self-coordination are the specific properties of correct parallel algorithms.

The verification of each of these properties represents a non-trivial combinatorial problem. Let us consider some of them.

As shown in [7-9], the verification of the cyclic logical control algorithm A is, to a great extent, reduced to the analysis of the corresponding α-net $\alpha(A)$ formed by label pairs $\mu_i \rightarrow \nu_i$ and by the value N_0. Of great importance in this analysis is the verification of safety and liveliness of the net $\alpha(A)$. The notion of α-net safety was defined above, and we shall call an α- net live (without departing from the terminology adopted in the Petri net theory) when any transition in any sequence of transitions can take place again (some time later).

It will be natural to draw the analogy between such properties of the algorithm A as irredundancy and recoverability, on the one hand, and liveliness of the net $\alpha(A)$, on the other, and then to try to reduce the verification of the first two properties to that of the latter. It should be borne in mind, however, that for some algorithms such reduction will not be complete, since the net $\alpha(A)$ does not contain data on informational interactions between the chains of the algorithm A. Actually, there exist irredundant and recoverable algorithms with corresponding unlive α-nets and, on the other hand, there exist live α-nets corresponding to redundant or unrecoverable algorithms.

One can speak more definitely about the relationship between the algorithm self-coordination property and the safety of the corresponding α-net.

Theorem 2 *If the net $\alpha(A)$ is safe, then the algorithm A is self-coordinated.*

In any case, the verification of algorithm correctness is, to some degree, reduced to that of liveliness and safety of the α- net $\alpha(A)$. The verification of liveliness is known to be somewhat easier and may be performed by the method suggested for ordinary Petri nets. This method reduces the problem to the solution of logical equations [10,11]. It is more difficult to verify the safety property. The direct method of integrated verification of these two qualities is known to be applicable to α-nets and Petri nets of a much wider class. This method involves the construction of a set of all reachable states and is practically realizable only for small nets. More promising for the purpose seem to be the reduction methods using local simplification operations, which sequentially decrease the dimensionality of the net being analysed [12,8]. In case of α-nets, we may confine ourselves to a pair of such operations: loop removal and substitution [13].

Loop Removal Operation. The transition τ_i is removed from the net if $\mu_i = \nu_i$ and the net contains another transition τ_j so that $\mu_i \subseteq \mu_j$ or $\nu_i \subseteq \nu_j$.

Substitution Operation. Let some unmarked (not intersecting with N_0) set of places π satisfies the following conditions for every transition τ_i:

1) if $\pi \cap \mu_i \neq \emptyset$, then $\pi = \mu_i$ and $\pi \cap \nu_i = \emptyset$,
2) if $\pi \cap \nu_i \neq \emptyset$, then $\pi \subseteq \nu_i$,
3) if $\pi = \mu_i$ and $\pi \subseteq \nu_j$, then $\nu_j \cap \nu_i = \emptyset$,

Then the set π and all the transitions τ_i, for which $\pi = \mu_i$, are eliminated; and each transition τ_j, for which $\pi \subseteq \nu_j$, is replaced by the set of transitions obtained from τ_j by substituting π for ν_i taken from the eliminated transitions.

Let, for example, $\pi = \{2, 3\}$ and $\pi \cap N_0 = \emptyset$, with places 2 and 3 come across only in the following transitions:

$$2.3 \to 5, \quad 2.3 \to 1.7, \quad 7.4 \to 2.3, \quad 4.8 \to 4.2.3$$

Then the substitution operation consists in replacing the given fragment by the following one:

$$7.4 \to 5, \quad 7.4 \to 1.7, \quad 4.8 \to 4.5, \quad 4.8 \to 4.1.7$$

Theorem 3 *α-net liveliness and safety are invariants of the transformations performed by the loop removal and substitution operations.*

The following theorem affirms the convergence of the reduction process.

Theorem 4 *The reduction of any live and safe α-net by means of loop removal and substitution operations can be carried on till its completion, i.e. till a net with a single transition $N_0 \to N_0$ is obtained.*

5 Partial state assignment

Both software and hardware implementation of PRALU-algorithms are possible. In the case of the latter the logic circuit synthesis is preceded by the "standardization" of the algorithm, i.e. by changing it for an equivalent parallel automaton and by coding its partial states.

Traditionally, when all the states are incompatible, each of them can be represented by a Boolean vector that can be implemented, for example, as a combination of flip-flop states. But in the case of a parallel automaton which can find itself in several partial states at the same time, this method is inadequate.

A trivial method of partial state assignment can be suggested. Let us take a special coding Boolean variable z_i for each partial state p_i (element of M) and assume that the automaton is in the state p_i if and only if $z_i = 1$. It follows that the transition $\mu_i \to \upsilon_i$ can be carried out in two steps: first, the value 0 is given to the variables z_j for which

$p_j \in \mu_i$; second, the value 1 is given to the variables z_k for which $p_k \in \upsilon_i$.

Though simple, this method has a drawback: the number of variables to be introduced for coding may appear to be unjustifiably large, and that impedes the subsequent hardware implementation. In order to facilitate the latter, one has to minimize the length of the code.

It was suggested in [14] that partial states can be coded by ternary vectors (with components 0,1 and -, where the symbol - is interpreted as an arbitrary value of the corresponding Boolean variable) which have to be non-orthogonal for parallel states. In this case the latter ones can be implemented by a single Boolean vector. For instance, if $1-0-0, -10--$ and $---10$ are the coding ternary vectors for three parallel states, then the vector 11010 does implement all of them.

In order to minimize the code length it is useful to take orthogonal vectors for any states which are not parallel (i.e. are incompatible). By that sometimes another good quality of the code can be reached: the quality of displacing. In that case the execution of any transition can be completely reduced to the transfer the automaton into the states forming the set $\upsilon_i \backslash \mu_i$; as to the states from $\mu_i \backslash \upsilon_i$, the automaton leaves them automatically. The code with such a property has been called the displacing ternary code (DT-code).

Assume that when constructing the DT-code we do not use the entire information contained in the set T of the transitions $\mu_i \rightarrow \upsilon_i$, but instead of that take into account only the set S of all global states (i.e., the sets of partial states which can exist simultaneously). In other words, we admit direct transitions between any elements of the set S. The DT-code that can be found under such restrictions has been called the universal one (UDT-code). But it can exist not for every parallel automaton.

Let us call a global state c-maximum if it corresponds to a maximum complete subgraph of the graph G, representing the relation of parallelism on the set of partial states.

Theorem 5 *The UDT-code for a parallel automaton exists iff every its global state is c-maximum.*

Let p_i be an arbitrary partial state, P_j is a global state, $c(p_i)$ and $c(P_j)$ correspondingly are their coding vectors. If p_i does not belong to P_j, then some state p_k can be found belonging to P_j and not parallel with p_i (otherwise the complete subgraph $G(P_j)$ corresponding to P_j would not be maximum). When assigning orthogonal coding vectors to non-parallel partial states we get $c(P_j)$ orthogonal with $c(p_i)$, so that the state p_i would be displaced any time when the global state P_j is realized. But if $G(P_j)$ is not maximum, then such a partial state p_i not belonging to P_j can be found, that is parallel with every partial state from P_j, so $c(p_i)$ would not be orthogonal with $c(P_j)$, and, consequently, p_i could not be displaced by P_j. ∎

Suppose, for instance, that some automaton with partial states 1,2,3,4,5,6 has the transitions $1 \to 2.3$, $2 \to 4.5$, $3.5 \to 6$, $4.6 \to 1$, from which it is not difficult to find all the global states $P_1 = \{1\}, P_2 = \{2,3\}, P_3 = \{3,4,5\}$ and $P_4 = \{4,6\}$. The following UDT-code can be found in that case:

$$\begin{array}{cc} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} z_1 \\ z_2 \end{array} & \left[\begin{array}{cccccc} 1 & 0 & 0 & - & 0 & 1 \\ 1 & 1 & - & 0 & 0 & 0 \end{array}\right] \end{array}$$

It gives rise to the coding vectors of the global states: $c(P_1) = 11$, $c(P_2) = 01, c(P_3) = 00$ and $c(P_4) = 10$. It is easy to verify that each of them is orthogonal to the coding vectors of the partial states not belonging to P_i: for instance $c(P_2)$ is orthogonal with $c(4) = -0$, etc.

A useful consequence of the theorem 5 can be mentioned.

If the condition of the theorem 5 is fulfilled, then there is no need in getting all the global states for construction the UDT-code: finding the relation of parallelism on the set of partial states is quite sufficient for that purpose.

It was proved by Hack in [5] that when EFC-net is live and safe, the set P of all places can be covered by the subsets of P having each exactly one common element with every reachable marking. This result can be used for proving the following

Theorem 6 *Every reachable marking of a live and safe EFC-net is* *c-maximum.*

Indeed, if some of those markings is not c-maximum, there exists a place compatible with all elements of this marking. From one side it has to belong to some of the mentioned subsets, from the other it has not, because each of them contains already one element of the marking and cannot contain another. This contradiction proves the theorem.

The next theorem follows, in its turn, from the previous two.

Theorem 7 *Assigning orthogonal ternary vectors to every pair of non-parallel partial states and non-orthogonal vectors to parallel ones, we get the UDT-code for every correct parallel automaton.*

The construction of the UDT-code becomes difficult if only we strive to find the minimal code, i.e. a code with minimal number of coding Boolean variables. The arising problems can be expressed and solved in terms of the graph theory.

Theorem 8 *The constructing of the minimal UDT-code for a correct parallel automaton A can be reduced to covering the graph $\bar{G}(A)$, supplementary to $G(A)$, with minimal number of complete bipartied subgraphs.*

In general case the problem of finding minimal UDT-codes is NP-hard. But there were proposed practically good algorithms solving it for small automata or when suboptimal solutions are admitted [14,15].

6 Concluding steps

Having obtained the UDT-code, it is easy to pass from every chain

$$\mu_i : -k_i' \rightarrow k_i'' \rightarrow \upsilon_i$$

to a simple sequent

$$k_1^* \vdash k_1^{**}$$

executing it. This is done as follows:

(1) The ternary vectors, presenting the result of “intersection” of the vectors coding the label components, are assigned to the labels μ_i and υ_i.

(2) The obtained vectors are interpreted as conjunction terms γ_i' and γ_i''.

(3) The sequent terms are found from the formulae

$$k_1^* = \gamma_i' \& k_i',$$
$$k_1^{**} = k_i'' \& \gamma_i''.$$

As an example, consider the chain

$$5.6 : -a\bar{c} \to \bar{u}vx \to 4.6.9.$$

Assume that the partial states forming the chain labels are coded by the sets of values of the variables z_1, z_2, z_3 and z_4 as follows:

$$\begin{array}{cc} & \begin{array}{cccc} 4 & 5 & 6 & 9 \end{array} \\ \begin{array}{c} z_1 \\ z_2 \\ z_3 \\ z_4 \end{array} & \begin{bmatrix} - & 1 & - & 0 \\ 1 & - & 1 & - \\ - & - & - & 0 \\ 1 & - & - & - \end{bmatrix} \end{array}$$

Then the vector 11– is assigned to the initial chain label, and the vector 0101 is assigned to the terminal one. On the whole, the chain is implemented by the simple sequent

$$z_1 z_2 a\bar{c} \vdash \bar{u}vx\bar{z}_1\bar{z}_3 z_4.$$

Therefore, having coded the partial parallel automaton states, we shall obtain a system of simple sequents:

$$k_1^* \vdash k_1^{**}, i = 1 \ldots n$$

which can be implemented in some or other element basis.

Suppose we shall confine ourselves to the basis of programmable logic arrays (PLA), with RS-flip-flops used as storage elements which fix the values of coding variables. In this basis, the system of simple sequents is implemented in a rather easy way: the conjunction terms k_1^* and k_1^{**} are represented by ternary vectors, and the system as a whole is represented by a pair of ternary matrices which are further interpreted as tuning matrices for two PLA stages, a conjunctive and a disjunctive ones [6].

This simple solution is however feasible only for comparatively small systems of sequents "accommodated" on one PLA. In general case, one has to construct a logic circuit composed of several PLAs. It is necessary for that to decompose a large system of Boolean functions given in the sum-of-products form into a set of similar systems of limited dimensions.

Some methods of solving this problem based on an application of matrix logical equations have been suggested in [6,18].

References

[1] Zakrevskij A.D. To the theory of parallel algorithms of logical control (Russian). - Izvestija AN SSSR. Technicheskaja kibernetika, 1989, No 5, p.179-191.

[2] Peterson J.L. Petri net theory and the modeling of systems. - N.J., 1981.

[3] Zakrevskij A.D., Vasiljonok V.K. The formal description of logical control algorithms for designing discrete systems (Russian). - Electronnoje modelirovanie, 1984, v.6, No 4, p.79-84.

[4] Zakrevskij A.D. Parallel automaton (Russian). - Doklady AN BSSR, 1984, v.28, No 8, p.717-719.

[5] Hack M.T. Analysis of production schemata by Petri nets. - Project MAC TR-94. Cambridge, 1972.

[6] Zakrevskij A.D. Logic synthesis of cascade circuits (Russian). - Moscow, Nauka, 1981.

[7] Zakrevskij A.D. The analysis of concurrent logic control algorithms. - Lecture Notes in Computer Science, v.278, Spinger- Verlag, 1987, p.497-500.

[8] Zakrevskij A.D. The reduction method for verification of the correctness of parallel logical control algorithms (Russian). - Doklady AN BSSR, 1983, v.27, No 7, p.617-619.

[9] Zakrevskij A.D. On the correctness of parallel algorithms of logical control (Russian). - Izvestija AN SSSR. Technicheskaja kibernetika, 1987, No 4, p.106-112.

[10] Sifakis J. The computing of traps in the condition-action systems (Russian). - In: The theory of discrete control devices. Moscow, Nauka, 1982, p.182-190.

[11] Zakrevskij A.D. To the checking of ordinary Petri nets liveliness (Russian). - Doklady AN BSSR, 1985, v.29, No 11, p.1006- 1009.

[12] Berthelot C., Roucairol C. Reduction of Petri nets. - Lecture Notes in Computer Science. Springer Verlag, 1976, v.45, p.202-209.

[13] Zakrevskij A.D. The checking of correctness of parallel logical control algorithms (Russian). - Programmirovanije, 1987, No 5, p.31-35.

[14] Zakrevskij A.D. The execution of parallel logical control algorithms on programmed logic arrays (Russian). - Avtomatika i telemechanika, 1983, No 7, p.116-123.

[15] Zakrevskij A.D. Block coding of partial states of automata performing the concurrent logical control algorithms (Russian). - Izvestija AN SSSR. Technicheskaya kibernetika, 1983, No 5, p.3-11.

[16] Zakrevskij A.D. State assignment problem in hardware implementation of parallel algorithms of logical control. Preprint No 16. - Minsk, Institute of Engineering Cybernetics. 1993, 11p.

[17] Zakrevskij A.D. On minimizing the coding matrix of partial states (Russian). - Doklady AN BSSR, 1993, v.37, No 6, pp.8-10.

[18] Bochmann D., Zakrevskij A.D., Posthoff Ch. Boolesche Gleichungen. - Berlin, VEB Verlag Technik, 1984.

Arkadij Zakrevskij,
Institute of Engineering Cybernetics,
Surganova str. 6,
220012 Minsk, Belarus,
e-mail: zakr@newman.basnet.minsk.by

Received 13 December, 1995