ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

PROGRESSIVE INFORMATION TECHNOLOGIES

UDC 004.94 : 004.2

MULTIPROCESS IMPLEMENTATION OF ALGORITHMS FOR ALGEBRAIC SYNTHESIS OF A FINITE STATE MACHINE

Babakov R. M. – Dr. Sc., Associate Professor, Professor of the Information Technologies Department, Vasyl' Stus Donetsk National University, Vinnytsia, Ukraine. ROR: https://ror.org/00xfvkq36. ORCID: https://orcid.org/0000-0001-7196-0912.

Barkalov A. A. – Dr. Sc., Professor, Professor of the Institute of Computer Science and Electronics, University of Zielona Gora, Zielona Gora, Poland. ROR: https://ror.org/04fzm7v55. ORCID: https://orcid.org/0000-0002-4941-3979.

Titarenko L. A. – Dr. Sc., Professor, Professor of the Institute of Computer Science and Electronics, University of Zielona Gora, Zielona Gora, Poland. ROR: https://ror.org/04fzm7v55. ORCID: https://orcid.org/0000-0001-9558-3322.

ABSTRACT

Context. The problem of the parallel implementation of two algorithms for algebraic synthesis of a finite state machine with datapath of transitions is considered. This type of state machine can be used as an alternative to a finite state machine with a canonical structure in order to reduce hardware expenses in the state machine circuit. The object of the study is algorithms for finding complete and partial solutions to the problem of the algebraic synthesis of a finite state machine, which have a parallel implementation based on a mechanism if processes. The first of these algorithms is the known algorithm for the complete sequential enumeration of state encoding variants with a fixed set of transition operations. The second algorithm implements an infinite enumeration of state encoding variants based on pseudo-random encoding. The goal of each algorithm is to find a solution to the algebraic synthesis problem with as few uncovered transitions as possible in a given time. This paper proposes an approach to increasing the speed of these algorithms, which consists in their parallel implementation using all processor cores available on the computer. This contributes to finding more efficient solutions to the algebraic synthesis problem in a given time, which can lead to lower hardware expenses in the device circuit.

Objective. Multiprocess implementation and research of algorithms for finding solutions to the algebraic synthesis problem of finite state machine with datapath of transitions.

Method. The research is based on the structure of a finite state machine with a datapath of transitions. The synthesis of the state machine circuit is preceded by a stage of algebraic synthesis, the result of which is the combination of a certain method of state encoding with the assignment of certain arithmetic-logical operations to individual transitions of the finite state machine. Such a combination is a solution to the problem of algebraic synthesis of a finite state machine with a datapath of transitions. In the general case, for a given finite state machine there are many solutions, each of which can be either complete (each transition is covered by one of the given operations) or partial (when part of the transitions remains uncovered by any of the operations). The more transitions in a partial solution are covered, the less hardware expenses are spent on implementing the state machine circuit and the better the solution found. The search for the best solutions requires an enumeration of a large number of possible variants of state encoding. In this case, it does not matter in principle whether the enumeration of the encoding variants is carried out sequentially or in a pseudo-random manner. In this work, in order to speed up the search for solutions to the algebraic synthesis problem, a parallel implementation of two algorithms is proposed, for which the "multiprocessing" module of the Python language is used. Both algorithms (the sequential search algorithm and the pseudo-random search algorithm) were implemented in programmatic way and investigated on the example of an abstract control algorithm using an i5-13500 processor. The purpose of the experiments was to evaluate the improvement of solutions to the algebraic synthesis problem found in the same running time by single-process and multi-process implementations of the specified algorithms.

Results. Using the example of an abstract control algorithm, it is demonstrated that, in general, multiprocess implementation of the considered algebraic synthesis algorithms allows finding better solutions to the algebraic synthesis problem in the same time than with a single-process (non-parallel) implementation of these algorithms. The advantage of the parallel implementation of the algorithms is preserved when using different sets of transition operations.

Conclusions. The algebraic synthesis of a finite state machine with a datapath of transitions is based on an algorithm for finding solutions to the algebraic synthesis problem. The paper proposes modified versions of previously known algorithms for finding such solutions, based on the use of the "multiprocessing" module of the Python language. The software implementation of these algorithms has proven that such an approach is generally better than a single-process search for state encoding variants, since it allows finding better solutions (solutions with fewer uncovered transitions) in the same time. The disadvantage of the proposed algorithms can be considered the use of more computer resources, which can negatively affect energy consumption.

KEYWORDS: finite state machine, datapath of transitions, arithmetic and logical operations, algebraic synthesis algorithms, parallel realization, Python.

DOI 10.15588/1607-3274-2026-2-12

ABBREVIATIONS

FSM is a finite state machine;
DT is a datapath of transitions;
TO is a transition operation;
GSA is a graph-scheme of algorithm.

NOMENCLATURE

A is a set of FSM states;
X is a set of logical conditions;
Y is a set of microoperations;
M is a number of FSM states;
L is a number of logical conditions analyzed by FSM;
K is a number of microoperations formed by FSM;
R is a bit capacity of state code;
O is a set of transitions operations;
O_i is an element of set O;
I is a number of TO;
B is a number of uncovered FSM transitions;
P is a number of logical processors;
a_i is a some FSM state;
G is a graph-scheme of control algorithm;
N is a number of possible state encoding variants;
U_{A1} is a number of uncovered transitions (algorithm A1);
U_{A2} is a number of uncovered transitions (algorithm A2);
U_{A3} is a number of uncovered transitions (algorithm A3);
U_{A4} is a number of uncovered transitions (algorithm A4);
E_{1-3} is an efficiency of parallel implementation of algorithm A3 compared to algorithm A1;
E_{2-4} is an efficiency of parallel implementation of algorithm A4 compared to algorithm A2;
E_{1-4} is an efficiency of parallel implementation of algorithm A4 compared to algorithm A1.

INTRODUCTION

The control unit CU is an essential component of a modern digital system and performs function of coordinating the operation of all system blocks [1–3]. There are various structural models of the CU, such as a finite state machine, a microprogram control unit, a compositional microprogram control unit, etc. [4–6]. A finite state machine is usually used in circuits where the critical characteristic is speed. An FSM circuit allows for state machine transitions that depend on several logical conditions simultaneously to be performed in one clock cycle of the device. This maximizes the speed of the circuit, but at the same time maximizes hardware expenses for its implementation. An increase in hardware expenses, in turn, worsens such characteristics of the circuit as cost, energy consumption, reliability, size, etc.

The desire to reduce hardware expenses in an FSM circuit has led to the emergence of a large number of methods and approaches that take into account both the structural organization of the FSM and the element base used [4, 7–10]. One such approach is the principle of operational transformation of FSM state codes, according to which state codes receive a scalar interpretation and are considered not only as binary vectors, but also as decimal numbers in various formats [11]. This allows transforming state codes during the execution of FSM transitions using arithmetic-logical operations, such as addition, subtraction, conjunction, bitwise shift, etc. Such operations are commonly called transition operations. Each TO can be assigned a circuit characterized by certain hardware expenses.

For a given FSM, the state codes can be chosen so that the same TO allows to implement several FSM transitions. In this case, the TO's circuit remains unchanged, and hardware expenses for its implementation remain fixed. On the contrary, in an FSM with a canonical structure, each additional transition leads to an increase in hardware expenses in the circuit that implements the transition function.

The principle of operational transformation of state codes generates the structure of an finite state machine with a datapath of transitions [11, 12]. In FSM with DT structure, a set of TO circuits is considered as a datapath (operational unit) that processes the code of the current state of the FSM under the control of operation selection signals. If some of the transitions were not implemented using a given set of TOs, they are implemented in a canonical way using a system of Boolean equations. Its combinational circuit within DT is considered as a separate TO that implements its own subset of FSM transitions. Such a combinational circuit will be absent if all the transitions of the FSM are implemented (covered) by a given set of arithmetic-logical operations.

Before synthesizing the circuit of an FSM with DT, algebraic synthesis of the FSM should be carried out [12, 13]. It involves encoding states and establishing a correspondence between the set of TOs and FSM transitions. These actions should be mutually consistent and contribute to using as few TOs as possible, which would cover as many transitions as possible. The only known way to perform algebraic synthesis is a complete search of state encoding variants for a given set of TOs [14]. The disadvantage of this method is the large amount of time spent on the analysis of even a simple FSM (10–20 states), which complicates the practical application of FSM with DT.

This paper proposes an approach to increasing the speed of algebraic synthesis, which consists in parallelizing the algorithm proposed in [14]. Also considered is the parallelization of the algorithm in which the state codes of an FSM are not selected sequentially, but in a pseudo-random manner. The proposed approach allows for a relatively simple increase in the speed of algebraic synthesis approximately as many times as the number of computer processor cores involved in algebraic synthesis. The implementation of parallel algorithms is performed in the Python language using the "multiprocessing" module.

DOI 10.15588/1607-3274-2026-2-12

The object of the study is the process of searching for complete and partial solutions to the problem of algebraic synthesis of a finite state machine using specialized state coding algorithms.

The subject of the study is parallel algorithms for searching for complete and partial solutions to the problem of algebraic synthesis of an FSM with DT, based on complete sequential search or pseudo-random generation of the state code of the FSM.

The purpose of the work is to develop and study the effectiveness of parallel algorithms for algebraic synthesis of a finite state machine with datapath of transitions, built on the basis of the "multiprocessing" module of the Python programming language.

1 PROBLEM STATEMENT

Let the FSM with DT be given in the form of a graph-scheme of the algorithm [10, 15, 16]. According to the GSA, the following are formed: a set of states $A=\{a_1,...,a_M\}$, a set of input signals $X=\{x_1,...,x_L\}$, a set of microoperations $Y=\{y_1,...,y_K\}$ and a certain number of FSM transitions B. The set of transition operations $O=\{O_1,...,O_I\}$ is also given. Each element $O_i \in O$ represents an arithmetic-logical operation, which involves a certain interpretation of the binary codes of the FSM states. GSA and the set O are the input data for the algebraic synthesis of the FSM with DT.

It is necessary to investigate the FSM synthesis algorithms A1 – A4 by the way of determine $E_{1-3}=U_{\text{A1}}-U_{\text{A3}}$, $E_{2-4}=U_{\text{A2}}-U_{\text{A4}}$, $E_{1-4}=U_{\text{A1}}-U_{\text{A4}}$, where $U_{\text{A}_i}=f_i(O,\,t)$, $t\in\{1,\,2,\,3,\,4,\,5\}$ min.

2 REVIEW OF THE LITERATURE

The variety of methods for optimizing FSM is widely covered in the literature [3, 6, 10]. The structure and process of synthesizing of an FSM with DT for a given GSA, are presented in [11, 12]. In [13] it is proposed to represent the result of algebraic synthesis as a composition of two components: the variant of state encoding and the correspondence of TO to FSM transitions. Such a result is called a formal solution to the algebraic synthesis problem, since it allows synthesizing a finite state machine according to the structure of FSM with DT [12].

In [14], the concept of a formal solution is detailed depending on the number of state machine transitions covered by operations from a given set of TOs. If all transitions are covered by the given TOs, the formal solution is called complete. If at least one transition remains uncovered (i.e., cannot be implemented using any of the given TOs for the selected state encoding variant), the formal solution is called partial.

Partial solutions to the algebraic synthesis problem should not be considered worse than full solutions. The effectiveness of a partial solution is determined only by hardware expenses for implementing the circuit of the FSM with DT according to this solution. A full solution may require a larger number of TOs or a set of TOs with more complex circuits than a partial solution, which uses a smaller number of simpler TOs, and the number of uncovered transitions is only a few percent. The final verdict on the choice of a partial formal solution can be made only after synthesizing the FSM circuit in a given element basis using specialized CAD with obtaining numerical values of the amount of hardware expenses.

To study the effectiveness of a particular method of algebraic synthesis, it is sometimes convenient to keep the set of TOs fixed. The only aspect that remains variable is the method of state encoding – that is, the correspondence between the states of the FSM and the set of admissible state codes. According to [14], the number of such variants is determined by expression (1). When the set of TOs is fixed, this expression also specifies the number of possible formal solutions to the algebraic synthesis problem for a given FSM

$$N=\frac{(2^R)!}{(2^R-M)!}. \qquad (1)$$

For a 10-state FSM, $N=29\cdot10^9$. Synthesis the circuit using Xilinx Vivado CAD takes several minutes for each of these solutions. Therefore, under normal conditions, it is impossible to synthesize the FSM circuit for all cases and obtain a numerical estimate of the hardware expenses. To compare the solutions with one another, the following rule can be proposed: the solution that results a smaller number of uncovered transitions is preferable. The validity of this rule stems from the fact that, given a fixed set of TOs, the hardware expenses for implementing the circuits of these TOs remain constant regardless of the number of covered transitions. However, a larger number of uncovered transitions requires a more complex combinational circuit to implement them in a canonical way.

We will also assume that two solutions with the same number of uncovered transitions are considered equivalent. This assumption does not account for differences in the canonical implementation of uncovered transitions, in particular, the minimization of the system of Boolean equations. However, given the typically small number of uncovered transitions, such differences can be neglected.

We will also consider all complete solutions, if any, to be identical. These assumptions allow us to formulate a strategy for constructing algorithms to search for formal solutions in the following way:

– if a complete solution is found, the algorithm can terminate, since other complete solutions (if they are found) will not lead to a reduction in the hardware expenses in the FSM circuit;

– among the solutions found during the algorithm's allotted time, the best one is which yields the minimum number of uncovered state machine transitions.

DOI 10.15588/1607-3274-2026-2-12

This strategy forms the basis of the algorithm proposed in [14]. The essence of the algorithm is to sequentially search through all possible variants of state encoding, the number of which is determined by expression (1). For each encoding variant, the number of uncovered transitions is calculated using the method proposed in [13]. The algorithm terminates in two cases: when a complete solution is found or when the time allotted for the algorithm to work is exhausted. In the latter case, the algorithm produces the best of the partial solutions found – that is, the solution with the minimum number of uncovered transitions.

Studies conducted in [13, 14] have shown that the use of the Python language results in low performance for such algorithms, although it simplifies their software implementation. Increasing the number of formal solutions considered within a given time will make it possible to find solutions with fewer uncovered transitions and to reduce hardware expenses in the circuit of FSM with DT.

3 MATERIALS AND METHODS

As a basis, let us consider the algorithm for deriving partial solutions to the algebraic synthesis problem by means of an exhaustive search of state-encoding variants. Its generalized form is shown in Fig. 1.

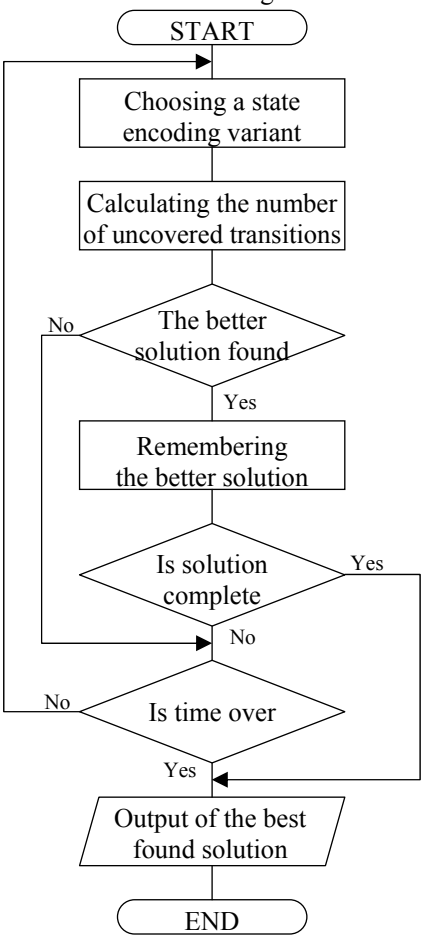

Figure 1 – Generalized block diagram of the algorithm for finding complete and partial solutions to the algebraic synthesis problem

This algorithm makes it possible to obtain the best partial solution within the allotted time. If a complete solution is found during the search, it is considered the best of all possible solutions, and the algorithm terminates the subsequent search.

In Fig. 1, the first block of the algorithm implements the method of the state encoding. Let us consider two possible methods.

Method 1. State codes are selected by sequentially searching through all possible variants. This method was proposed in [14]. Its advantage is that the algorithm is able to consider all possible ways of state encoding and choose the best of the possible formal solutions (complete or partial with the minimum number of uncovered transitions). However, such a search is effective only under the condition of an unlimited running time of the algorithm, which is usually not feasible. With a limited running time, the algorithm for different sets of TOs will always review only a subsequence of state encoding variants, while most of the variants will never be reviewed.

Method 2. The codes of the FSM states are chosen in a pseudo-random manner at each iteration of the algorithm. With this method, the search for state encoding variants is uncontrolled and chaotic. Regardless of how long the algorithm runs, some encoding variants may remain unexamined, and some variants may be examined multiple times. We also lose the ability to perform a sequential search for individual parts (for example, perform half of the search on one day, the other half on another). The advantage of this method is that it operates over the entire range of state encoding variants.

Without conducting appropriate studies, it is impossible to say which of the methods is capable of finding better solutions for an arbitrary GSA within the same time. Let us make the following assumptions. The symbol A1 will denote the algorithm shown in Fig. 1, which uses method 1 to select state encoding variants. Accordingly, the algorithm that uses method 2 will be denoted by A2.

To accelerate the execution of algorithms A1 and A2, it is proposed to develop and investigate their parallel implementations. The processors of modern computers have a multi-core architecture, in which each core contains two logical processors. Modern operating systems treat logical processors as independent cores, distributing the tasks performed among them. Increasing the performance of algorithms A1 and A2 is possible provided that their workload is rationally distributed across the logical processors involved. The efficiency of logical professors depends on the architecture of the cores and the implementation of the multithreading mechanism in a specific processor model, as well as on the task-scheduling algorithms of the operating system.

In Python, the use of logical processors is possible through the “multiprocessing” module [17–19]. Architecturally, a multiprocess program in Python consists of a main process and a certain number of child processes. The main process can exchange data with the child processes, distributing tasks among them and collecting the results of their work. It should be noted that the alternative

DOI 10.15588/1607-3274-2026-2-12

"threading" module in Python does not implement true multithreading, executing all threads on one physical core (it is actual for Python version 3.12.0) [19, 20].

Below is a fragment of the program code that demonstrates the launch of processes for parallel execution of algorithms A3 and A4. To ensure maximum program performance, it was decided to abandon the process manager in favor of using global variables based on the *Value* class. Processes based on the *permutation* function implement a parallel sequential search of state encoding variants. Processes based on the *random_codes* function (commented out in the program) implement a parallel search of pseudo-random state-encoding variants.

```
# Number of processors available:
P = mp.cpu_count()

# Shared minimal number
# of uncovered transitions:
n_min = mp.Value("i", G.p)

# Shared counter of checked
# state encoding variants:
total_permut = mp.Value("i", 0)

t = 300 # Timelimit of proccesses execution

LOCK = mp.Lock()

procs = []

time1 = time.time() # Start time

for i in range(0, P):

    # Serial enumeration
    proc = mp.Process(target=permutation,
           args=(i, n_min, t, G_pic,
           O_pic, LOCK, total_permut))

    # Random enumeration
    #proc = mp.Process(target=random_codes,
           args=(i, n_min, t, G_pic,
           O_pic, LOCK, total_permut))

    # Adding process to process list
    procs.append(proc)

    # Starting process
    for proc in procs:
        proc.start()

    # Waiting for processes finishing
    for proc in procs:
        proc.join()

time2 = time.time() # End time

print(int(time2-time1), " seconds passed.")

print("Minimal number of uncovered
      transitions:", n_min.value)
print("Done after", total_permut.value,
     "permutations.")
```

Let us analyze the features of parallelization of algorithm A1. The duration of the algorithm is influenced by the number of iterations, which is determined by expression (1). Parallelization of the algorithm on P logical processors consists of dividing the entire number of iterations into P equal parts, so that each logical processor performs a specific portion of the sequential searches. At the start of execution, the main process sends the child processes a "starting point" for sequential searches, after which the child processes operate independently, using shared data structures to store the best solution found and to monitor the remaining computation time. After the child processes complete their work, the main process displays the most optimal solution on the screen (i.e., the state-encoding variant that yields the minimum number of uncovered transitions) and terminates the execution.

Let us analyze the features of parallelization of algorithm A2. Since there is no sequential search in this algorithm, there is no need to provide individual initial data to the child processes. Each child process, after creation, works independently within the allotted time. At each iteration, it generates a pseudo-random variant of state encoding, calculates the number of uncovered transitions, and updates information about the best solution found in the data structure shared by all child processes. The main process launches the child processes, waits for their completion, and then displays the most optimal solution found on the screen.

The considered features enabled the authors to develop software implementations of algorithms A1 and A2 based on the Python "multiprocessing" module. The purpose of the program development was not the practical application of the algorithms, but a comparative study of their efficiency. In this regard, a simplification was introduced into algorithm A1. Instead of evenly dividing the entire range of state encoding variants among the computer cores, each child process, upon starting, generates a pseudo-random variant of state encoding once, from which it subsequently performs a sequential search of variants until the end of its execution. This simplification is due to the fact that even with a small number of states (from 20 to 30) the value of N becomes so large that the probability of intersection of the ranges of encoding variants generated randomly is close to zero. In this way, the search range of each child process is almost guaranteed not to overlap with the ranges of other child processes.

To generate successive variants of states encoding, the *permutations* function of the "itertools" module is used. To generate a pseudo-random variant of states encoding, the list of 2^R permissible state codes is shuffled using the *shuffle* function of the "np.random" module, after which the first M elements of the list are selected for states $a_0, ..., a_{M-1}$ encoding. The work with child processes is implemented using the "multiprocessing" module. The development was carried out using Python version 3.12.0.

DOI 10.15588/1607-3274-2026-2-12

4 EXPERIMENTS

The purpose of the experiments is to compare the efficiency of single-process and multi-process implementations of proposed algorithms. We will consider four algorithms:

– algorithm A1, which does not use parallelization;

– algorithm A2, which does not use parallelization;

– algorithm A1, which uses parallelization (hereafter denoted as algorithm A3);

– algorithm A2, which uses parallelization (hereafter denoted as algorithm A4).

The procedure for conducting the experiment is as follows:

1. The GSA is given.

2. The set of TOs is given.

3. Each of the algorithms A1 – A4 is run with the same time constraint.

4. The best solution is determined based on the results of each algorithm, i.e., the solution with the minimum number of uncovered transitions. Let U_{A1}, U_{A2}, U_{A3} and U_{A4} be the minimum numbers of uncovered transitions obtained by algorithms A1, A2, A3 and A4, respectively.

5. The efficiency E_{1-3} of the parallel implementation of algorithm A1 is determined by the difference $(U_{A1} - U_{A3})$. Although this value is not relative, it allows the designer to assess whether it is advisable to use a larger number of processor cores to achieve E_{1-3} additional covered transitions.

6. The efficiency E_{2-4} of the parallel implementation of the algorithm A2 is determined by the difference $(U_{A2} - U_{A4})$.

According to points 1 and 2 of the procedure for conducting the experiment, the values of E_{1-3} and E_{2-4} depend on the input data, which consists of a given GSA and a given set of TOs. For a more substantiated study, it is advisable to repeat the experiment several times with different input data. To vary the input data, one should choose either another GSA or another set of TOs. In this work, we repeat the experiment for a single GSA and different sets of TOs, since this approach is more easily implemented at the software level.

Let an FSM with DT be represented by an abstract GSA G (Fig. 2). This GSA does not contain information about the output function, since the operational transformation of state codes concerns only the transition function. The GSA is marked by the states of the Moore finite state machine: the initial and final vertices are marked by the state a_0, and each operator vertex is marked by a separate state from a_1 to a_{19}. The main characteristics of the GSA are $M = 20$, $B = 27$, and $R = 5$.

As TOs, we will choose arithmetic-logical operations that are performed on the current state code and a certain constant. In arithmetic operations, the state code and the constant are interpreted as unsigned integers, in logical operations, they are interpreted as binary vectors. Each operation has its own constant, which lies in the range $[0; 2^R - 1]$ of permissible state codes. The result of any TO is mapped to the same range by retaining only the R lowest binary bits and discarding all higher bits (if any are generated during the operation).

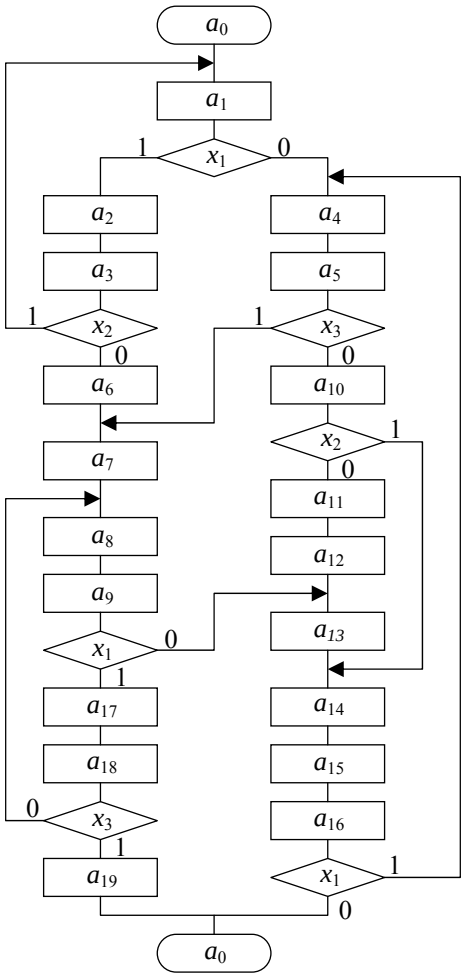

Figure 2 – Graph-scheme of algorithm G

In short, the transition operations will be denoted as follows: “+5”, “–2”, “×8”, “÷4”, “&10110”, “∨10011”, “⊕11101”, etc. Decimal constants in the process of circuit implementation will be represented in binary R-bit unsigned format.

Each of the studied algorithms A1 – A4 searches for partial solutions to the algebraic synthesis problem. An example of a partial solution is shown in Fig. 3. The solution is obtained using a set of four transition operations: “+3”, “+7”, “–4”, “⊕10011”. In the figure, each vertex corresponding to the FSM state contains a state code represented as a decimal number and its binary equivalent. Transitions covered by the given TOs are denoted by the corresponding TO, which converts the code of the output state into the code of the input state. Transitions not covered by any transition operation are marked with “-----”.

DOI 10.15588/1607-3274-2026-2-12

As can be seen, the GSA in Fig. 3 contains 12 uncovered transitions, which in the process of FSM circuit synthesis, must be implemented in a canonical manner using the corresponding system of Boolean equations. This solution cannot be considered the best, since increasing the algorithm's running time or changing the set of TOs used can potentially yield solutions with a smaller number of uncovered transitions.

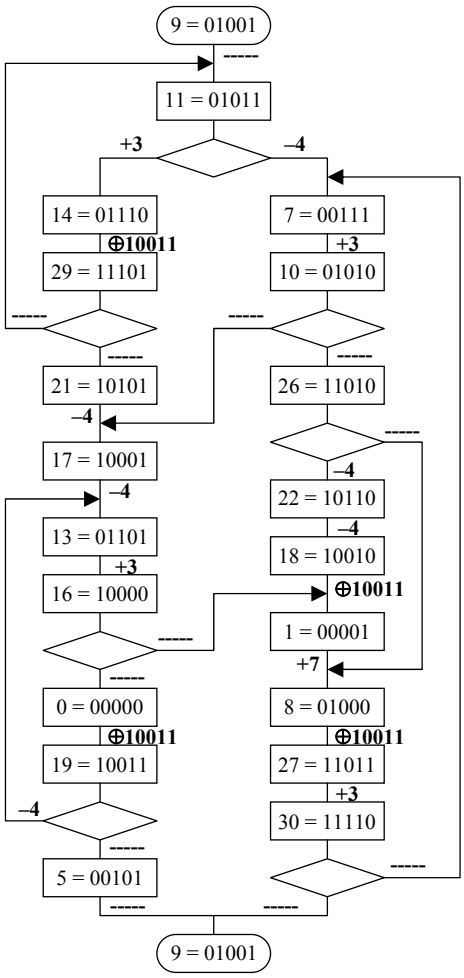

Figure 3 – Example of partial solution for GSA *G* with 12 uncovered transitions

The results produced by the software implementation of algorithms A1 – A4 are in text form and contain information about the set of used TOs and the state encoding variant. The solution shown in Fig. 3 has the following text form:

```
Operations: +3  +7  -4  ^19
a0=9 a1=11 a2=14 a4=7 a3=29 a6=21 a5=10
a7=17 a10=26 a8=13 a9=16 a13=1 a17=0
a14=8 a11=22 a12=18 a15=27 a16=30 a18=19
a19=5
Uncovered transitions: 12
```

This solution does not contain information about the correspondence between operations and individual transitions. However, such correspondence can be obtained manually. For example, if the operations "+3", "+7", "–4", "⊕10011" are used (the latter is denoted as "^19" according to Python notation), the transition from the state a_{15} with code $27_{10} = 11011_2$ to the state a_{16} with code $30_{10} = 11110_2$ can be performed only using the "+3" operation. However, none of the given TOs can be used to perform the transition from the state a_0 with code $9_{10} = 01001_2$ to the state a_1 with code $11_{10} = 01011_2$, so this transition is implemented in a canonical way.

Thus, the results of the developed program allow us to determine the number of uncovered transitions based on the results of algorithms A1 – A4. Let us consider the results of applying these algorithms to the test GSA *G* (Fig. 2) and to different sets of TOs. All experiments were carried out on a computer with an i5-13500 processor and Windows 11 OS. When the parallel algorithms A3 and A4 were executed, all 20 available logical processors were used.

5 RESULTS

Table 1 shows the results of the study of algorithm A1. For ten different sets of four TOs, indicated in the columns $O_1 - O_4$, the obtained values U_{A1} were found during different execution times of the algorithm (from 1 to 5 minutes).

Table 1 – Values U_{A1} for different sets of TOs for different execution times of algorithm A1

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	15	15	15	15	15
2	+2	–7	+11	&18	20	18	18	18	18
3	+1	–1	+2	^30	20	20	20	20	20
4	+3	–1	×2	÷4	17	15	15	15	15
5	+3	+5	^24	÷2	18	18	18	18	18
6	+9	–15	^19	^26	21	20	20	20	20
7	+7	–10	×4	&18	18	18	18	18	18
8	+1	+18	–5	–22	16	16	15	15	15
9	^7	^21	^26	^30	17	17	16	16	16
10	+1	–6	^22	∨18	17	17	16	16	16

Table 2 shows the results of the study of algorithm A2 using the example of GSA *G*. The values U_{A2} were obtained under the same conditions as the values U_{A1} in Table 1.

Tables 3 and 4 show the results of the study of algorithms A3 and A4, respectively. The experimental conditions are the same as those used for the study of algorithms A1 and A2.

DOI 10.15588/1607-3274-2026-2-12

Table 2 – Values U_{A2} for different sets of TOs for different execution times of algorithm A2

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	13	13	13	13	13
2	+2	–7	+11	&18	13	13	13	12	12
3	+1	–1	+2	^30	13	13	13	13	12
4	+3	–1	×2	÷4	13	12	12	12	12
5	+3	+5	^24	÷2	12	12	12	11	11
6	+9	–15	^19	^26	13	13	13	12	12
7	+7	–10	×4	&18	13	13	13	13	13
8	+1	+18	–5	–22	13	13	12	12	9
9	^7	^21	^26	^30	13	13	13	13	13
10	+1	–6	^22	∨18	13	13	13	13	13

Table 3 – Values U_{A3} for different sets of TOs for different execution times of algorithm A3

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	14	14	14	14	14
2	+2	–7	+11	&18	16	16	16	16	16
3	+1	–1	+2	^30	14	14	14	14	14
4	+3	–1	×2	÷4	15	15	15	15	15
5	+3	+5	^24	÷2	15	15	15	15	15
6	+9	–15	^19	^26	15	14	14	14	14
7	+7	–10	×4	&18	16	16	16	16	16
8	+1	+18	–5	–22	14	14	14	14	14
9	^7	^21	^26	^30	15	15	15	13	13
10	+1	–6	^22	∨18	15	15	15	13	13

Table 4 – Values U_{A4} for different sets of TOs for different execution times of algorithm A4

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	12	11	10	10	10
2	+2	–7	+11	&18	12	12	12	12	12
3	+1	–1	+2	^30	13	12	12	11	11
4	+3	–1	×2	÷4	12	12	11	11	11
5	+3	+5	^24	÷2	12	12	12	12	12
6	+9	–15	^19	^26	12	12	11	11	11
7	+7	–10	×4	&18	12	12	11	11	11
8	+1	+18	–5	–22	12	12	12	11	11
9	^7	^21	^26	^30	12	12	12	11	11
10	+1	–6	^22	∨18	11	11	11	11	11

For each execution of algorithms A1 or A2, about 55 million state-encoding variants were processed within 5 minutes. For each execution of algorithms A3 or A4, about 445 million variants were processed, that is, 8 times more. This ratio is due solely to the architecture of the i5-13500 processor used, which contains 6 high-performance cores and 8 low-performance cores. Although the processor cores support 20 threads (logical processors), the Intel Hyper-Threading technology used in it does not guarantee a 20-fold increase in performance and demonstrates different efficiency for different tasks. We can only state that parallelizing algorithms A1 and A2 using the Python "multiprocessing" module on the i5–13500 processor made it possible to increase their performance by about a factor of 8.

Let's analyze the results.

1. Determine the efficiency $E_{1–3}$ of the parallel implementation of algorithm A1. Let's construct table 5, each element of which is defined as the difference between the values in the corresponding cells of Tables 1 and 3.

Table 5 – Efficiency $E_{1–3}$

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	1	1	1	1	1
2	+2	–7	+11	&18	4	2	2	2	2
3	+1	–1	+2	^30	6	6	6	6	6
4	+3	–1	×2	÷4	2	0	0	0	0
5	+3	+5	^24	÷2	3	3	3	3	3
6	+9	–15	^19	^26	6	6	6	6	6
7	+7	–10	×4	&18	2	2	2	2	2
8	+1	+18	–5	–22	2	2	1	1	1
9	^7	^21	^26	^30	2	2	1	3	3
10	+1	–6	^22	∨18	2	2	1	3	3

According to Table 5, the following generalized conclusions can be drawn:

1) Parallelization of algorithm A1 makes it possible to obtain better solutions to the algebraic synthesis problem within the same time. The values in Table 5 show how many FSM transitions were additionally implemented in an operational way thanks to algorithm A3 compared to algorithm A1. Note that this gain in the number of covered transitions was obtained on the same processor within the same time. Since all processor cores are used during the execution of algorithm A3, its execution leads to greater power consumption by the computer.

2) Increasing the execution time of the algorithms does not lead to an increase in the number of covered transitions. For example, in row 7, algorithm A3 made it possible to find two additional covered transitions in the first minute, after which it was unable to improve this result during the next four minutes. This indicates that it usually makes little sense to run the A3 algorithm for too long, expecting a significant improvement in the results.

3) The set of used TOs significantly affects the comparative results of algorithms A1 and A3. If, for the fourth set of operations, algorithm A3 provides almost no advantage over algorithm A1, the third and sixth rows demonstrate a significant increase in the number of covered transitions.

Averaging the values in Table 5, it can be noted that algorithm A3 makes it possible to cover approximately 2.7 more transitions than the algorithm A1 within the specified time. This is 10 % of the total number of transitions $B = 27$.

2. Determine the efficiency $E_{2–4}$ of parallel implementation of algorithm A2. Construct Table 6, each element of which is defined as the difference between the values in the corresponding cells of Tables 2 and 4.

DOI 10.15588/1607-3274-2026-2-12

Table 6 – Efficiency E_{2-4}

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	1	2	3	3	3
2	+2	–7	+11	&18	1	1	1	0	0
3	+1	–1	+2	^30	0	1	1	2	2
4	+3	–1	×2	÷4	1	0	1	1	1
5	+3	+5	^24	÷2	0	0	0	–1	–1
6	+9	–15	^19	^26	1	1	2	1	1
7	+7	–10	×4	&18	1	1	2	2	2
8	+1	+18	–5	–22	1	1	0	1	–2
9	^7	^21	^26	^30	1	1	1	2	2
10	+1	–6	^22	∨18	2	2	2	2	2

Table 7 – Efficiency E_{1-4}

N	Transition operations				Execution time, min.				
	O_1	O_2	O_3	O_4	1	2	3	4	5
1	+1	+4	–5	^23	3	4	5	5	5
2	+2	–7	+11	&18	8	6	6	6	6
3	+1	–1	+2	^30	7	8	8	9	9
4	+3	–1	×2	÷4	5	3	4	4	4
5	+3	+5	^24	÷2	6	6	6	6	6
6	+9	–15	^19	^26	9	8	9	9	9
7	+7	–10	×4	&18	6	6	7	7	7
8	+1	+18	–5	–22	4	4	3	4	4
9	^7	^21	^26	^30	6	6	5	6	6
10	+1	–6	^22	∨18	6	6	5	5	5

According to Table 6, the conclusions are as follows:

1) The values E_{2-4} are generally lower than those in Table 5. On average, algorithm A4 allows covering 1.2 additional transitions, which is close to 4.5 % of the total number of transitions of the GSA G. Nevertheless, any number of additionally covered transitions helps to reduce hardware expenses in the FSM circuit. Therefore, in the author's opinion, using algorithm A4 as an alternative to algorithm A2 is advisable (provided that the additional power consumption by the computer processor is ignored).

2) The negative values in some cells of the table indicate that algorithm A4, for the corresponding set of operations, could not find better solutions compared to algorithm A2. It should be noted that both algorithms use pseudo-random state encoding at each of their iterations. Therefore, there may be situations in which a larger number of considered state-encoding variants does not lead to better solutions. Given the rarity of such situations, they cannot be considered a drawback of algorithm A4.

3) As in the case of Table 5, increasing the execution time of algorithm A4 does not generally lead to a significant increase in E_{2-4}, i.e., to obtaining better solutions. Perhaps, when the algorithm's execution time is increased to several hours, the result of comparing the A4 and A2 algorithms will be more pronounced. However, such an increase in time is unpleasant from a practical point of view, since it delays the design process of devices based on FSM with DT.

Note that all considered algorithms A1, A2, A3, and A4 are alternatives to each other, since they solve the same problem. According to the values in Tables 1 – 4, the least optimal algorithm is A1, and the most optimal is A4. By analogy with the efficiencies E_{1-3} and E_{2-4}, we will calculate the efficiency E_{1-4} of algorithms A1 and A4. To do this, we will perform element-by-element subtraction of the values between Tables 1 and 4. The obtained result is presented in Table 7.

The contents of Table 7 demonstrate that the use of algorithm A4, compared to algorithm A1, allows covering, within the same program execution time, an additional 6 FSM transitions on average, which amounts to 22% of the total number of transitions of GSA G.

The inefficiency of algorithm A1 is due to the fact that at each run it performs a sequential search of the state-encoding variants, even though it starts the search from a randomly generated state-encoding variant. Within 5 minutes of program execution, the algorithm manages to examine only a very small portion of all possible state-encoding variants. As a result, most of the state codes remain unchanged during the search, and only the codes of the last few states are modified. This limits the attempts to use the given TOs to cover transitions between states whose codes do not change. Consequently, a smaller number of transitions are covered by the given TOs.

Algorithm A4 generates a random set of codes for all FSM states in each iteration. This allows the program to cover different ranges of state-encoding variants during execution and to try to apply the given TOs to the entire set of transitions. Studies have shown that such a strategy is more efficient compared to a completely sequential search, as well as to a sequential search starting with a pseudo-random state-encoding variant.

6 DISCUSSION

Each of the algorithms A1 – A4 studied in this work contains a pseudo-random component. This means that different runs of the same program for the same GSA and the same set of TOs can yield different results. For example, in Table 2, the number 9 in the last column of the eighth row is the smallest among all values in Tables 1 – 4. This value indicates that algorithm A2, at the fifth minute of execution, happened to find a state-encoding variant that allowed covering 18 of the 27 transitions of GSA G, leaving only 9 transitions uncovered. It should be understood that repeated runs of algorithm A2 with the same set of TOs will not necessarily produce the same result. On the other hand, such results – or even better ones – may be obtained by chance when using any of the four algorithms for any set of TOs.

In general, the scope of experimental study is sufficient to determine the advantages and disadvantages of each algorithm under investigation. The main

DOI 10.15588/1607-3274-2026-2-12

conclusion of the research is that the approach used to search for state-encoding variants significantly affects the quality of the solutions obtained for the problem of algebraic synthesis of FSM with DT. In addition to the considered sequential and pseudo-random approaches to searching for state-encoding variants, other approaches may also be explored. One such approach, for example, may involve an additional search for the TOs being used.

CONCLUSIONS

The article proposes new algorithms for algebraic synthesis of a finite state machine with a datapath of transitions. Their key feature is the use of a task-parallelization mechanism on multi-core processors. The goal of developing these algorithms is to increase the number of iterations of FSM state-encoding variants within a given time. This helps to find solutions to the problem of algebraic synthesis of FSM with DT that have a fewer uncovered transitions and lead to reduced hardware expenses in the FSM circuit.

The scientific novelty of this work lies in the fact that, for the first time, algorithms for finding solutions to the algebraic synthesis problem using process-based parallelization technology have been developed, implemented and investigated. Studies of the test GSA have shown that the new algorithms are able to find significantly more efficient solutions within the same time as their prototype algorithms without parallelization. This confirms the relevance of their development and the feasibility of using them in the synthesis of finite state machines.

The practical use of the obtained results is possible in the development of methods and algorithms for the synthesis of finite state machines with operational transformation of state codes as part of specialized CAD tools for digital control systems.

Prospects for further research include solving a number of scientific and practical problems related to optimizing the proposed algorithms in order to obtain better solutions to the algebraic synthesis problem. The authors also consider it relevant to study the efficiency of the algorithms on large-sized GSAs, including pseudo-randomly generated ones. This will make it possible to more clearly demonstrate the effectiveness of the corresponding methods of algebraic synthesis of FSM with DT.

ACKNOWLEDGEMENTS

The work was carried out as an initiative research within the scientific direction of the Department of Information Technologies of Vasyl' Stus Donetsk National University.

DECLARATIONS

Conflict of interest: The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship, or otherwise, that could affect the research and its results presented in this paper.

Authors' contributions: Roman Babakov: the algorithms A1–A4; Larysa Titarenko: software implementation of the algorithms; Alexander Barkalov: experimental research of the algorithms efficiency.

Data availability: The manuscript has no associated data.

Software availability: The manuscript has no associated software.

Use of artificial intelligence tools: The authors confirm that they did not use artificial intelligence technologies in creating the submitted work.

REFERENCES

1. Bailliul J., Samad T. Encyclopedia of Systems and Control. Springer. London, UK, 2015, 1554 p. DOI: https://doi.org/10.1007/978-1-4471-5058-9
2. Czerwinski R., Kania D. Finite state machines logic synthesis for complex programmable logic devices. Berlin, Springer, 2013, 172 p. DOI: https://doi.org/10.1007/978-3-642-36166-1
3. Baranov S. Logic and System Design of Digital Systems. Tallin, TUTPress, 2008, 267 p.
4. Micheli G. D. Synthesis and Optimization of Digital Circuits. McGraw-Hill, Cambridge, MA, USA, 1994, 579 p.
5. Minns P., Elliot I. FSM-Based Digital Design Using Verilog HDL. JohnWiley and Sons. Hoboken, NJ, USA, 2008, 408 p. DOI: https://doi.org/ 10.1002/9780470987629
6. Skliarova I., Sklyarov V., Sudnitson A. Design of FPGA-based circuits using hierarchical finite state machines. Tallinn, TUT Press, 2012, 240 p.
7. Klimovich A. S., Solov'ev V. V. Minimization of Mealy finite-state machines by internal states gluing. *Journal of Computer and Systems Science International,* 2012, Volume 51, pp. 244–255. DOI: https://doi.org/10.1134/S1064230712010091
8. Grout I. Digital Systems Design with FPGAs and CPLDs. Elsevier Science, Amsterdam, The Netherlands, 2008, 784 p. DOI: https://doi.org/10.1016/B978-0-7506-8397-5.X0001-3
9. Kubica M., Opara A., and Kania D. Technology Mapping for LUT-Based FPGA. *Lecture Notes in Electrical Engineering,* Vol. 713, Springer, Cham, 2021, 207 p. DOI: https://doi.org/10.1007/978-3-030-60488-2
10. Baranov S. Logic Synthesis for Control Automata. Dordrecht, Kluwer Academic Publishers, 1994, 312 p.
11. Barkalov A. A., Babakov R. M. Operational formation of state codes in microprogram automata, *Cybernetics and Systems Analysis,* 2011, Volume 47 (2), pp. 193–197. DOI: https://doi.org/10.1007/s10559-011-9301-y
12. Barkalov A. A., Titarenko L. A., Babakov R. M. Synthesis of Finite State Machine with Datapath of Transitions According to the Operational Table of Transitions, *Radio Electronics, Computer Science, Control,* 2022, Volume 3 (62), pp. 109–119. DOI: https://doi.org/10.15588/1607-3274-2022-3-11
13. Barkalov A. A., Babakov R. M. A Matrix Method for Detecting Formal Solutions to the Problem of Algebraic Synthesis of a Finite-State Machine with a Datapath of Transitions, *Cybernetics and Systems Analysis,* 2023, Volume 59 (2), pp. 190–198. DOI: https://doi.org/10.1007/s10559-023-00554-6
14. Babakov R. M., Barkalov A. A., Titarenko L. A., Voitenko M. O. Algorithmic Differences of Complete and Partial Algebraic Synthesis of a Finite State Machine with

DOI 10.15588/1607-3274-2026-2-12

Datapath of Transitions. *Radio Electronics, Computer Science, Control,* 2024, Volume 4, pp. 143–152. DOI: https://doi.org/10.15588/1607-3274-2024-4-14
15. Kubica M., Kania D., Kulisz J. A Technology Mapping of FSMs Based on a Graph of Excitations and Outputs. *IEEE Access,* 2019, Volume 7, pp. 16123–16131. DOI: https://doi.org/10.1109/ACCESS.2019.2895206
16. Baranov S. Finite State Machines and Algorithmic State Machines. Seattle, WA, USA: Amazon, 2018, 185 p.
17. Zaccone G. Python Parallel Programming Cookbook. Packt Publishing, 2015, 286 p.
18. Palach J. Parallel Programming with Python. Packt Publishing, 2014, 122 p.
19. Nelli F. Parallel and High Performance Programming with Python. AVA, 2023, 392 p.
20. Nguyen Q. Mastering Concurrency in Python. Packt Publishing, 2018, 446 p.

Received 08.12.2025.
Accepted 10.04.2026.
Published 26.06.2026.

УДК 004.94 : 004.2

МУЛЬТИПРОЦЕСОВА РЕАЛІЗАЦІЯ АЛГОРИТМІВ АЛГЕБРАЇЧНОГО СИНТЕЗУ МІКРОПРОГРАМНОГО АВТОМАТА

Бабаков Р. М. – д-р техн. наук, доцент, професор кафедри інформаційних технологій Донецького національного університету імені Василя Стуса, м. Вінниця, Україна. ROR: https://ror.org/00xfvkq36. ORCID: https://orcid.org/0000-0001-7196-0912.

Баркалов О. О. – д-р техн. наук, професор, професор Інституту комп'ютерних наук та електроніки університету Зеленогурського, м. Зельона Гура, Польща. ROR: https://ror.org/04fzm7v55. ORCID: https://orcid.org/0000-0002-4941-3979.

Тітаренко Л. О. – д-р техн. наук, професор, професор Інституту комп'ютерних наук та електроніки університету Зеленогурського, м. Зельона Гура, Польща. ROR: https://ror.org/04fzm7v55. ORCID: https://orcid.org/0000-0001-9558-3322.

АНОТАЦІЯ

Актуальність. Розглянуто задачу паралельної реалізації двох алгоритмів алгебраїчного синтезу мікропрограмного автомата з операційним автоматом переходів. Цей тип автомата може використовуватись в якості альтернативи мікропрограмному автомату з канонічною структурою з метою зменшення апаратурних витрат в схемі автомата. Об'єктом дослідження є алгоритми пошуку повних і часткових розв'язків задачі алгебраїчного синтезу автомата, що мають паралельну реалізацію на основі механізму процесів. Першим із цих алгоритмів є відомий алгоритм повного послідовного перебору варіантів кодування станів при фіксованій множині операцій переходів. Другий алгоритм реалізує нескінчений перебір варіантів кодування станів на основі псевдовипадкового кодування. Метою кожного з алгоритмів є пошук розв'язку задачі алгебраїчного синтезу з якомога меншою кількістю непокритих переходів за встановлений час. В даній роботі пропонується підхід до збільшення швидкодії цих алгоритмів, який полягає у їх паралельній реалізації із використанням усіх процесорних ядер, доступних на комп'ютері. Це сприяє пошуку більш ефективних розв'язків задачі алгебраїчного синтезу за встановлений час, які можуть приводити до менших апаратурних витрат в схемі пристрою.

Мета. Мультипроцесова реалізація і дослідження алгоритмів пошуку розв'язків задачі алгебраїчного синтезу мікропрограмного автомата з операційним автоматом переходів.

Метод. В основу дослідження покладено структуру мікропрограмного автомата з операційним автоматом переходів. Синтезу схеми автомата передує етап алгебраїчного синтезу, результатом якого є сполучення певного способу кодування станів із заставленням певних арифметико-логічних операцій окремим переходам автомата. Таке поєднання являє собою розв'язок задачі алгебраїчного синтезу мікропрограмного автомата з операційним автоматом переходів. У загальному випадку для заданого автомата існує безліч розв'язків, кожен з яких може бути як повним (кожен перехід покривається однією із заданих операцій) або частковим (коли частина переходів залишається непокритою жодною із операцій). Чим більше переходів у частковому розв'язку є покритими, тим менше апаратних ресурсів витрачається на реалізацію схеми автомата і тим кращим є знайдений розв'язок. Пошук кращих розв'язків потребує перебору великої кількості можливих варіантів кодування станів. При цьому немає принципового значення – чи перебір варіантів кодування здійснюється послідовно, чи у псевдовипадковий спосіб. В даній роботі з метою пришвидшення пошуку розв'язків задачі алгебраїчного синтезу запропоновано паралельна реалізація двох алгоритмів, для якої використано модуль «multiprocessing» мови Python. Обидва алгоритми (алгоритм послідовного перебору і алгоритм псевдовипадкового перебору) були реалізовані програмно і досліджені на прикладі абстрактного алгоритму керування із використанням процесора i5-13500. Метою експериментів була оцінка покращення розв'язків задачі алгебраїчного синтезу, знайдених за однаковий час роботи однопроцесовою і мультипроцесовою реалізаціями зазначених алгоритмів.

Результати. На прикладі абстрактного алгоритму керування продемонстровано, що загалом мультипроцесова реалізація розглянутих алгоритмів алгебраїчного синтезу дозволяє за однаковий час знайти кращі розв'язки задачі алгебраїчного синтезу, ніж однопроцесова (непаралельна) реалізація цих алгоритмів. Перевага паралельної реалізації алгоритмів зберігається при використанні різних наборів операцій переходів.

Висновки. В основі алгебраїчного синтезу мікропрограмного автомата з операційним автоматом переходів лежить алгоритм пошуку розв'язків задачі алгебраїчного синтезу. В роботі запропоновано модифіковані версії раніше відомих алгоритмів пошуку таких розв'язків, основані на використанні модуля «multiprocessing» мови Python. Програмна реалізація цих алгоритмів довела, що такий підхід в загальному випадку є кращим за однопроцесовий перебір варіантів кодування станів, оскільки дозволяє знайти кращі розв'язки (розв'язки з меншою кількістю операційно нереалізованих переходів) за той самий час. Недоліком запропонованих алгоритмів можна вважати застосування більшої кількості ресурсів комп'ютера, що може негативно впливати на його енергоспоживання.

КЛЮЧОВІ СЛОВА: мікропрограмний автомат, операційний автомат переходів, арифметико-логічні операції, алгоритм алгебраїчного синтезу, паралельна реалізація, Python.

DOI 10.15588/1607-3274-2026-2-12

ЛІТЕРАТУРА

1. Bailliul J. Encyclopedia of Systems and Control / J. Bailliul, T. Samad. – Springer : London, UK, 2015. – 1554 p. DOI: https://doi.org/10.1007/978-1-4471-5058-9
2. Czerwinski R. Finite state machines logic synthesis for complex programmable logic devices / R. Czerwinski, D. Kania. – Berlin : Springer, 2013. – 172 p. DOI: https://doi.org/10.1007/978-3-642-36166-1
3. Baranov S. Logic and System Design of Digital Systems / S. Baranov. – Tallin : TUTPress, 2008. – 267 p.
4. Micheli G. D. Synthesis and Optimization of Digital Circuits / G. D. Micheli. – McGraw-Hill : Cambridge, MA, USA, 1994. – 579 p.
5. Minns P. FSM-Based Digital Design Using Verilog HDL / P. Minns, I. Elliot. – JohnWiley and Sons : Hoboken, NJ, USA, 2008. – 408 p. DOI: https://doi.org/10.1002/9780470987629
6. Skliarova, I. Design of FPGA-based circuits using hierarchical finite state machines / I. Skliarova, V. Sklyarov, A. Sudnitson. – Tallinn : TUT Press, 2012. – 240 p.
7. Klimovich A.S. Minimization of Mealy finite-state machines by internal states gluing / A. S. Klimovich, V. V. Solov'ev // Journal of Computer and Systems Science International. 2012. – Volume 51. – P. 244–255. DOI: https://doi.org/10.1134/S1064230712010091
8. Grout I. Digital Systems Design with FPGAs and CPLDs / I. Grout. – Elsevier Science: Amsterdam, The Netherlands, 2008. – 784 p. DOI: https://doi.org/10.1016/B978-0-7506-8397-5.X0001-3
9. Kubica M. Technology Mapping for LUT-Based FPGA / M. Kubica, A. Opara and D. Kania // Lecture Notes in Electrical Engineering. – 2021. – Vol. 713. – Springer, Cham. – 207 p. DOI: https://doi.org/10.1007/978-3-030-60488-2
10. Baranov S. Logic Synthesis for Control Automata / S. Baranov. – Dordrecht : Kluwer Academic Publishers, 1994. – 312 p.
11. Barkalov A. A. Operational formation of state codes in microprogram automata / A. A. Barkalov, R. M. Babakov // Cybernetics and Systems Analysis. – 2011. – Volume 47 (2). – P. 193–197. DOI: https://doi.org/10.1007/s10559-011-9301-y
12. Barkalov A. A. Synthesis of Finite State Machine with Datapath of Transitions According to the Operational Table of Transitions / A. A. Barkalov, L. A. Titarenko, R. M. Babakov // Radio Electronics, Computer Science, Control. – 2022. – Volume 3 (62). – P. 109–119. DOI: https://doi.org/10.15588/1607-3274-2022-3-11
13. Barkalov A. A. A Matrix Method for Detecting Formal Solutions to the Problem of Algebraic Synthesis of a Finite-State Machine with a Datapath of Transitions / A. A. Barkalov, R. M. Babakov // Cybernetics and Systems Analysis. – 2023. – Volume 59 (2). – P. 190–198. DOI: https://doi.org/10.1007/s10559-023-00554-6
14. Algorithmic Differences of Complete and Partial Algebraic Synthesis of a Finite State Machine with Datapath of Transitions / [R. M. Babakov, A. A. Barkalov, L. A. Titarenko, M. O. Voitenko] // Radio Electronics, Computer Science, Control. – 2024. – Volume 4. – P. 143–152. DOI: https://doi.org/10.15588/1607-3274-2024-4-14
15. Kubica M. A Technology Mapping of FSMs Based on a Graph of Excitations and Outputs / M. Kubica, D. Kania, J. Kulisz // IEEE Access. – 2019. – Volume 7. – P. 16123–16131. DOI: https://doi.org/10.1109/ACCESS.2019.2895206
16. Baranov S. Finite State Machines and Algorithmic State / S. Baranov. – Machines, Seattle, WA, USA: Amazon, 2018. –185 p.
17. Zaccone G. Python Parallel Programming Cookbook / G. Zaccone. – Packt Publishing, 2015. – 286 p.
18. Palach J. Parallel Programming with Python / J. Palach. – Packt Publishing, 2014. – 122 p.
19. Nelli F. Parallel and High Performance Programming with Python / F. Nelli. – AVA, 2023. – 392 p.
20. Nguyen Q. Mastering Concurrency in Python / Q. Nguyen. – Packt Publishing, 2018. – 446 p.

DOI 10.15588/1607-3274-2026-2-12