An Efficient Verification Algorithm for Parallel Controllers

K. Biliński, J. M. Saul, E. L. Dagless, J. Szajna †

Department of Electrical and Electronic Engineering,
University of Bristol, Bristol BS8 1TR, United Kingdom

†Department of Computer Engineering and Electronics,
Higher College of Engineering, 65-246 Zielona Gora, Poland

Abstract

A new algorithm for verifying the equivalence of parallel controller designs is presented along with its implementation. The controller is specified using a Petri net, and its implementation is given as a netlist. The reachability graph of the Petri net is generated and simultaneously the network is implicitly simulated. By exploiting information from the reachability graph a reduction of the time and memory needed for verification has been achieved.

1 Introduction

VLSI circuit design methodology can be divided into two main stages: synthesis and verification. Synthesis, which is usually done automatically, refers to the process of creating a circuit from its specification. It is possible that synthesis may introduce incorrect by construction results for a particular circuit. For that reason, independent, automatic, and trustworthy verification tools are needed. The process of determining whether a designed circuit matches what was specified is called implementation verification, which consists of behavioural, logic and layout verifications [5].

The verification of combinatorial circuits has received the majority of attention for a long time, and so there are many techniques available that can verify combinatorial circuits efficiently. The verification of sequential circuits is considerably more difficult. The simplest approach uses exhaustive simulation. Since the number of states that have to be tested grows exponentially with the number of inputs and storage elements, this technique is very time and memory consuming and can only be used for small circuits (less than 10 latches). However, new more suitable algorithms have recently been presented for finite state machines (FSMs) which operate on a state transition graph (STG) in order to verify designs. In [3] two new approaches to FSM verification were introduced. The main idea of the first one is that two STGs are extracted: one from the register–transfer–level specification and the other from the netlist implementation. While extracting the second STG, the use of don't care information from the first STG enables the reduction of the number of states and the number of edges in the second STG. Then a graph multiplication method is used to check the equivalence. In the second approach, the STG of one circuit is enumerated and simultaneously simulated on the other one. An extension of the second method is presented in [4], [5], [6]. The advantage of this approach is that only the current path of the STG of the first FSM is stored, rather than the entire STG. Other methods called symbolic STG traversal algorithms were initially presented in [2], [5]. Within the methods, which can be implemented efficiently using BDDs, an input as well as a state space are implicitly enumerated. These methods work best for data–path–type circuits, i.e. circuits in which almost any state can be reached from any other. However, for parallel controllers in which there are many parallel interacting processes, the above methods cannot be used efficiently.

A parallel controller may be specified using an interpreted Petri net. As shown in [9] there are some advantages in using a Petri net specification to synthesize parallel controllers, especially when circuit area and speed are considered. In this paper, a new efficient approach for verifying parallel controllers design is given and illustrated with examples. The method presented is the first known approach to implementation verification of parallel controllers that are specified using Petri nets.

The paper is organised as follows: basic definitions are presented in Section 2. The algorithm for parallel controller verification is described of Section 3. In Section 3.1 an algorithm for reachability graph generation is given. The method of implicit simulation is shown in Section 3.2. Experimental results are discussed in Section 4. Section 5 draws conclusions and makes suggestions for future work.

2 Preliminaries

A Petri net is a bipartite, weighted, directed graph, which has two types of nodes called *places*, represented by circles, and *transitions*, represented by bars or boxes. Directed arcs connect the places and the transitions, with some arcs leading from the places to the transitions and others vice versa. To each arc a weight, a positive integer represented by a label, is assigned, where the m–weight arc can be viewed as

the set of m parallel arcs. When $m = 1$, the label is usually omitted. A marking is an assignment of *tokens*, represented as black dots, to the places. The position and the number of tokens changes during the net execution. Formally a Petri net PN is defined as a 5-tuple [8]:

$$PN = (P, T, F, W, M_0)$$

where: $P = \{p_1, p_2, ..., p_m\}$ is a finite non-empty set of places; $T = \{t_1, t_2, ..., t_n\}$ is a finite non-empty set of transitions; $F \subseteq (P \mathrm{x} T) \cup (T \mathrm{x} P)$ is a finite non-empty set of arcs; $W : F \rightarrow \{1, 2, 3, ...\}$ is a weight function; $M_o : P \rightarrow \{0, 1, 2, ...\}$ is the initial marking; $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

The net execution is performed according to simple rule for transition enabling and firing [8]. A Petri net is said to be an ordinary Petri net if all of its arc weights are set to one.

The behaviour of a sequential circuit can be modelled using a finite state machine (FSM). When there are many parallel, interacting subprocesses to be controlled, using a finite state machine can be awkward, since each state has to control all of them. A better solution is to divide each of the global sequential states into a number of concurrently active local states, with each local state controlling a different subprocess. The main difference between a finite state machine and parallel controller is that whilst a finite state machine has only one state active at any time, a parallel controller can have several states active simultaneously.

For a synchronous parallel controller specification a high level Petri net is used, which is called an interpreted synchronous Petri net. A predicate may be attached to each transition, which is a Boolean function of the controller's input signals. Moore outputs are associated with places and Mealy outputs with transitions. Each place represents a local state of the controller, and so the global state of the controller is equivalent to the current marking of the net. Thus the net execution defines the controller behaviour, which can be represented using a reachability graph. New rules for transition enabling and firing are introduced. A transition is enabled when all of its input places are marked and its predicate, if present, is asserted. All transitions are synchronized by a global clock and so all enabled transitions fire simultaneously. An example of a Petri net representation of a parallel controller is shown in Figure 1.a.

A state, in general, is a symbol indicating the internal state of the circuit. Each state has a unique bit vector, known as the state code, which representing that state. A state with only 1's and 0's as bit values is called a minterm state. A cube state can have the values in the different bit positions 0, 1, or don't–care (–). The cube state therefore represents a group of minterm states [5]. The reset or initial state is the state to which the machine goes after a power–up. If a circuit consists of n simple memory elements (latches or flip–flops) the state space is 2^n. However for a particular circuit, especially for a large one, the number of states that can be reached from an initial state is significantly smaller. All states that can be reached from the initial state are called valid states,

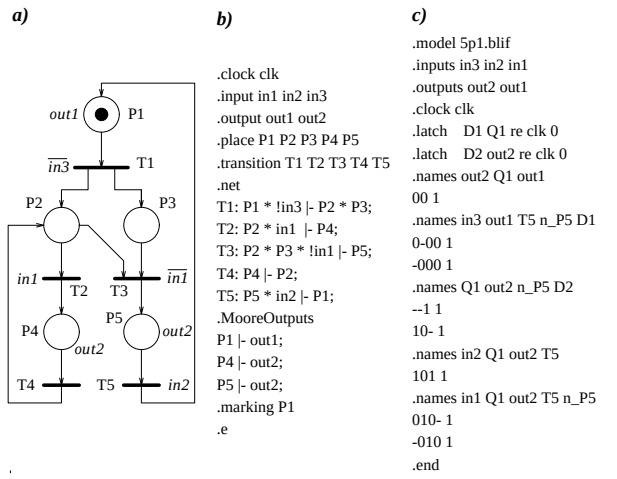

Figure 1: An example of input date formats; a) A Petri net specification of a controller, b) PNSF description of the net, c) BLIF representation of the controller's implementation.

while other states are called invalid. Two circuits are said to be equivalent if for all input sequences both circuits produce the same output sequence. To determine the equivalence of two circuits a correspondence between at least one state in each circuit should be found. Usually it is the initial state. This definition can be used for verifying circuits on a different level of description and/or to verify a specification of the circuit with its implementation.

During the design process, many descriptions of the same circuit are created, so that various verification steps should be used. In Figure 2, part of a typical synthesis pipeline and its connection with the proposed verification algorithm are given [5]. The problem of checking the equivalence of a behavioural description and a register–transfer–level description is referred to as behavioural verification which is a part of implementation verification. The logic verification, is the process of verifying the equivalence of two logic–level description of circuits — usually, the optimized and the unoptimized descriptions of the circuit.

The meaning of verification in this paper is as follows: for a given specification and its implementation the verification algorithm checks the correctness of a design. It is assumed that a parallel controller is synchronized using a global clock, has a reset state, is specified using an interpreted synchronous Petri net and the implementation is given as a netlist. An example of input date is shown in Figure 1.b[1] and 1.c[2].

[1] The Petri Net Specification Format (PNSF) allows a Petri net to be described in a textual form. The PNSF was developed at the University of Bristol[7].

[2] Berkeley Logic Interchange Format (BLIF) allow to describe a logic–level circuit specification, which consists of interconnected single–output gates and latches, in textual form. The BLIF was developed at the University of California, Berkeley [10].

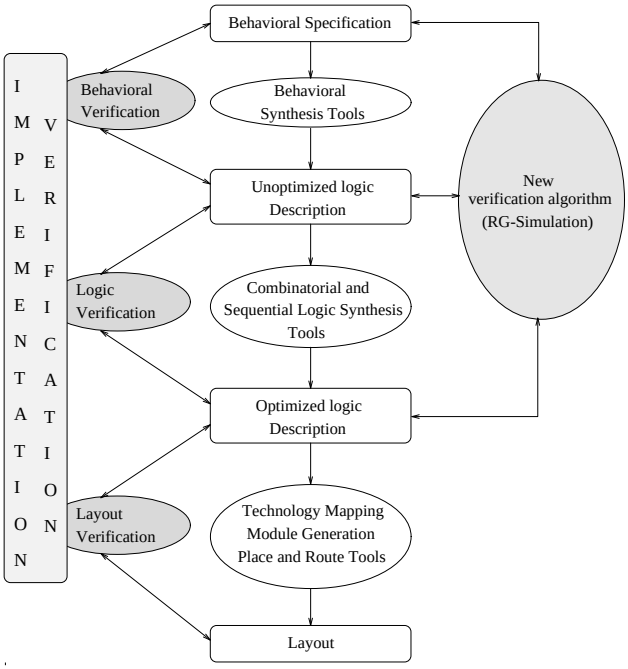

Figure 2: Part of a typical synthesis pipline.

```
generate_reachability_graph (M0 , RG)
{
/* Mo is the current node */
T = get_marking_transitions (M0);
/* Generate an InputVector for curren marking */
InputVector = NULL;
foreach_transition ( T , Ti ) {
    InputVector = InputVector + get_input_signals ( Ti );
}
/* Searching for a new node */
foreach_input_sequence ( InputVector, Ini ) {
    /* Create new node from Mo for input signal combination Ini */
    M1 = new_marking ( M0 , Ini );
    if ( M1 is new marking from M0 ) {
        if ( M1 is new node in reachability graph RG ) {
            /* Determine active outputs from simulated network */
            Out net = network_simulate ( Ini );
            /* Determine active outputs from reachability graph */
            Out graph = get_outputs ( M1 );
            if ( ! check_equivalence ( Out graph  Out net ) ) {
                return (FALSE);     /* Networks are not equivalent */
            }
            Add M 1 to reachability graph RG;
            if ( ! generate_reachability_graph ( M1, RG ) ) {
                return (FALSE);     /* Networks are not equivalent */
            }
        }
    }
}
return (TRUE);      /* Networks are equivalent */
```

Figure 3: Reachability graph generation algorithm.

3 Verification algorithm and its implementation

The main idea of the method presented here is to generate a reachability graph from the Petri net specification and simultaneously simulate the network. This approach is efficient for several reasons. First, the reachability graph of only one circuit (specification) has to be created and stored. Second, a cube input vector, which is computed in the reachability graph generation procedure, is placed into the simulation procedure, so that a smaller number of computation steps for simulation is required. The new approach is especially suited to circuits with many parallel processes (parallel controllers), and it is time and memory efficient, since the input space can be implicitly enumerated. The presented algorithm may be used for behavioural and logic verification.

The verification algorithm consists of two main parts: a reachability graph construction and simulation. The algorithm for reachability graph generation from a given Petri net specification of a circuit is presented first.

3.1 Reachability graph generation

To determine all of the valid states in the specified circuit, a reachability graph is generated. During the graph construction, all possible relations among the valid states are established. The algorithm is shown in Figure 3 and will be illustrated with an example.

Let us consider the net shown in Figure 4. At the beginning, the current state M_0 is the initial marking of the net. The initial marking is added to the reachability graph as the first node before calling the procedure **generate_reachability_graph()**. In this example the initial marking is 00100101, which becomes the first node in the graph. The routine first determines transitions that have all of their input places marked (transitions t_3 and t_6). Next an input vector is created. Since transitions t_3 and t_6 do not have predicates assigned the input vector is set to don't care (this means that the above transitions are enabled for any input signal combination). Next all possible combinations of the input signals are determined — a different subset of transitions may be fired for each combination. When all input signals are set to don't–care status only, one combination of transitions can fire. In such a case all of the enabled transitions fire simultaneously (here t_3 and t_6). Continuing the graph construction, for each firing sequence of transitions, the routine **new_marking()** tries to find a new marking. In the case described above only marking 01001001 is possible. Next the new marking M_1 is first checked to see whether it is a new marking which can be reached from M_0, and then M_1 is checked to determine if it is covered by any node that is already in the graph. If the marking does not appear in the graph it is added to the graph and the network simulation routine is called with the current input vector. A description of the simulation method is given in Section 3.2.

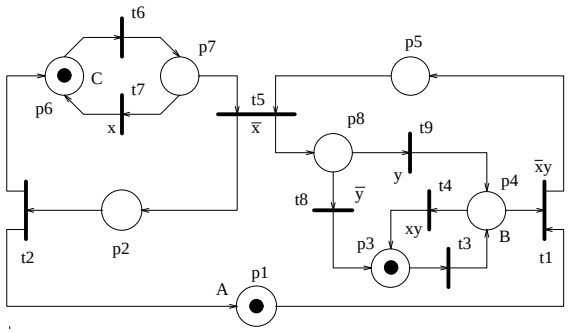

Figure 4: An example of a parallel controller specification.

It may seem redundant that first M_1 is checked to determine if it is a new node which can be reached from current marking M_0, and then also to check whether M_1 is a new node in the reachability graph, since if M_1 is a new node in the graph it is obviously enough to perform the rest of the algorithm. However, the search space for the first comparison is always considerably smaller than for the second one. Having the network simulated, the routine **check_equivalence()** is called to determine the equivalence relation between corresponding outputs. If for any pair of checked outputs different values appear, then the circuits are not equivalent. After equivalence has been determined, further node generation is performed by calling the routine **generate_reachability_graph()** recursively with M_1 as the input marking. These steps of the algorithm are illustrated in Figure 5.

3.2 Simulation

Whenever a new node of the reachability graph is produced by the routine **generate_reachability_graph()** an input vector is simulated on the circuit representing an implementation. When the input vector is a minterm the simulation is called minterm simulation; otherwise, it is called cube simulation [6]. Since the input vector in the presented algorithm is in general a cube and not a minterm, an algorithm for cube simulation is used. Usually, for the cube simulation a cube–splitting algorithm is performed on the input lines to produce a known value on the output and next–state lines [6]. However, in the presented algorithm (as shown in Section 3.1) valid input signals, i.e. signals whose value may change the present state of the controller, are always set to a known value. To determine the proper work of the simulator only a correct topological ordering of nodes within the simulated circuit is required (i.e. the nodes are ordered such that every node appears somewhere after all of its transitive fanin nodes).

Let us again consider the net of Figure 4. The controller which was synthesized from that specification and its BLIF description are shown in Figures 6.a and 6.b respectively. Node ordering during simulation and a function associated to each node are presented in Figure 6.c. Every function contains literals which either represent input signals or latches' outputs or are

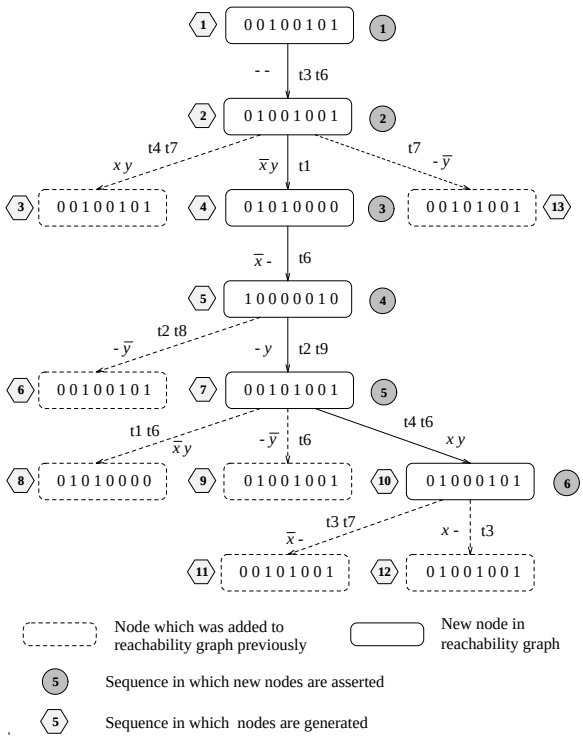

Figure 5: The sequence of node generation.

computed before being used in the function (nodes scheduled earlier in the node ordering). Now let us consider table of Figure 6.d which shows successive steps of simulation. The initial state of the controller is represented by step 1. Let us recall from the previous section that being in this state the controller changes its state for any combination of input signals, so that this input vector has all its input variables set to don't cares (lines x, y). In this case, taking the node ordering (Figure 6.c) into account we can compute value of every internal, next–state, and output node without considering value of input signals x, y. For each simulation step the routine **generate_reachability_graph()** guarantees that the input vector, although in general being a cube, has required input signals set to a known value.

When comparing the presented algorithm with algorithms for FSM verification, there are at least two reasons of efficiency of the new method. First, the generation of reachability graphs for parallel controllers does not require a cover of the ON and OFF sets of each primary output and next–state line to be generated. Second, it is not necessary to perform the cube–splitting algorithm, so that time and memory needed for the simulation is reduced.

4 Experimental results

In this section results that were achieved by using the algorithm presented above are described. The algorithm has been implemented in C and linked to the

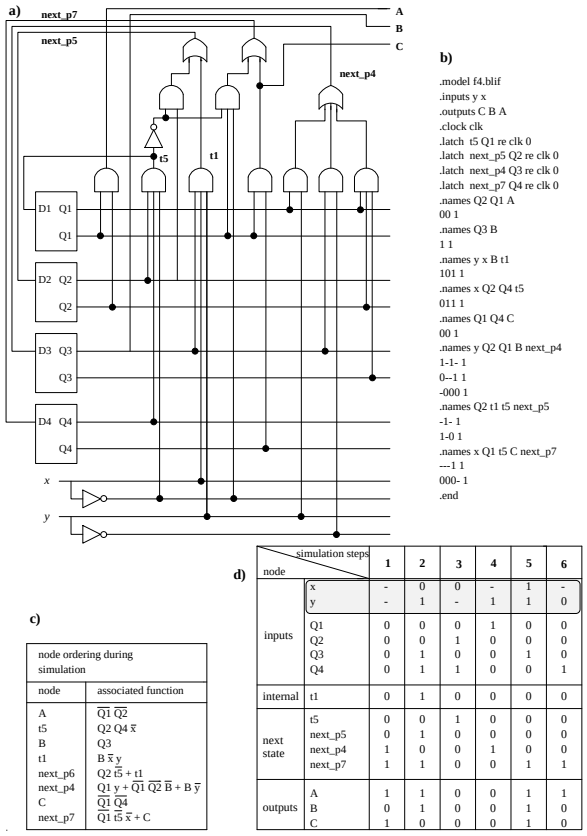

Figure 6: Simulation steps.

SIS sequential logic synthesis system [10]. It forms a part of the methodology that was developed for parallel controller design [1]. All examples were counted on a Sun SPARC–Station2 computer. For each of the examples verification time *Tcpu* and memory size needed *Memory* were considered. CPU times are quoted in seconds and memory required in kBs.

The algorithm was evaluated with three groups of parametrized benchmarks to find how its performance varies with problem size — based on the *sleeping barber*, *traffic light controller*, and *cigarette smokers* problems, respectively. The maximum size of benchmarks was limited to 20000 states or 100000 state transition edges in order to keep simulation time below one hour. In addition, to evaluate the verification results when real hardware designs are considered a set of examples, which were taken from published papers and various other sources, was also verified. Statistics of the benchmarks are given in Table 1. The first two columns show the number of primary inputs and primary outputs respectively. The next two columns show the number of places and transitions of the Petri net specification of each controller. The last two columns show the number of states and state transition edges of an equivalent state transition graph of each controller.

Each of the examples was encoded using a method presented in [9] (unoptimized description), and then

			Petri net		Equivalent STG	
Example	#in.	#out.	#pl.	#tr.	#st.	#ed.
barber2	3	2	10	11	18	72
barber3	4	2	14	15	81	486
barber4	5	2	18	19	324	2640
barber5	6	2	22	23	1215	13740
barber6	7	2	26	27	4374	67896
traffic2	0	3	10	8	8	8
traffic4	0	6	20	16	16	16
traffic8	0	12	40	32	34	34
traffic16	0	24	80	64	66	66
traffic32	0	48	160	128	130	130
smoke3	1	3	24	18	192	336
smoke4	1	4	32	24	331	662
smoke5	1	5	40	30	1222	4888
smoke6	1	6	48	36	4381	17524
smoke7	1	7	56	42	15316	30632
zigzag	6	11	20	18	40	80
fstore	8	8	18	19	48	86
pr_sp	2	1	21	22	39	65
react	10	9	16	13	29	129
fgen	9	4	13	14	80	846

Table 1: Statistics of examples: parallel and sequential controller representation.

synthesized using SIS (optimized description). Thus two logic–level descriptions of each of the examples were generated. Finally, using the algorithm presented here the behavioural and logic verification were performed by comparing the initial specification of an example with each of those two descriptions.

Since there are no standard benchmarks for verification, it is hard to compare the efficiency of the algorithm presented here with other techniques. A functionally equivalent sequential controller for each of the examples was also generated[3] Next, using the enumeration–simulation algorithm (verify_fsm) for sequential circuit verification [5], a set of corresponding results was obtained.

Table 2 summarizes results of the behavioural verification. The first two columns show the number of nodes and latches used to implemented each of the examples respectively. The next columns show statistics of verification when using the algorithm presented in this paper (pn_verify), and the verify_fsm approach. For the **barber** examples, the pn_verify approach is considerably better for all of the examples, considering both time and memory required. It can be seen that verification time for the pn_verify algorithm grows more quickly with the problem size than for verify_fsm. Using this observation we can assume that for bigger examples of this kind the verification time of verify_fsm approach is going to be shorter. However, the pn_verify approach remains considerably more memory efficient. When the **traffic** benchmarks are considered the comparison of equivalent results reveals significant improvements in terms of both verification time and memory needed when using the pn_verify algorithm. Additionally, in this case the verification time for pn_verify grows more slowly than for the verify_fsm approach. When the third set

[3] The method is based on converting a Petri net description into a single state–transition–graph, from which a sequential controller is next synthesized.

Example	#no.	#la.	pn_verify Tcpu (s)	pn_verify Mem. (kB)	verify_fsm Tcpu (s)	verify_fsm Mem. (kB)
barber2	45	6	0.7	362	2.1	624
barber3	64	9	1.3	492	6.0	836
barber4	85	11	6.7	522	31.2	1380
barber5	104	14	44.2	944	122.7	3070
barber6	124	16	380.2	1794	566.7	7372
traffic2	50	4	0.2	378	1.4	678
traffic4	94	7	0.4	422	4.4	814
traffic8	182	15	1.0	502	27.8	1150
traffic16	358	31	2.7	652	245.5	4224
traffic32	710	63	10.7	890	2290.8	13492
smoke3	120	14	1.7	602	13.5	974
smoke4	163	18	4.7	796	29.0	1108
smoke5	204	23	20.7	1538	86.4	1756
smoke6	240	27	81.6	4258	285.9	4096
smoke7	287	31	340.0	15066	1480.5	8496
zigzag	190	10	0.9	594	8.4	946
fstore	122	9	1.1	528	7.3	906
pr_sp	105	8	0.6	562	6.9	868
react	84	6	0.5	512	5.9	784
fgen	73	7	1.5	942	3.1	766

Table 2: Behavioural verification results.

Example	#no.	#la.	pn_verify Tcpu (s)	pn_verify Mem. (kB)	verify_fsm Tcpu (s)	verify_fsm Mem. (kB)
barber2	8	5	0.2	322	1.8	610
barber3	15	8	0.9	432	5.3	800
barber4	21	10	4.9	474	29.3	1280
barber5	30	13	37.6	824	114.0	2974
barber6	35	15	362.7	1408	520.7	6282
traffic2	9	4	0.1	312	1.3	644
traffic4	12	7	0.2	386	4.0	792
traffic8	40	15	0.6	476	24.9	1114
traffic16	82	31	1.9	588	237.8	3862
traffic32	157	63	4.9	794	2248.3	12964
smoke3	24	14	1.0	614	14.2	970
smoke4	34	18	2.4	782	29.4	1204
smoke5	42	23	11.5	1308	98.3	1936
smoke6	53	27	59.3	3068	497.3	4816
smoke7	59	31	295.4	9366	2013.5	9782
zigzag	56	10	0.7	584	8.2	938
fstore	42	9	0.8	464	5.9	882
pr_sp	16	8	0.2	498	6.5	800
react	26	6	0.2	442	5.3	712
fgen	14	7	1.2	558	2.9	666

Table 3: Logic verification results.

of benchmarks — **smoke** — is considered the verification time needed for pn_verify grows similarly to verify_fsm, but the verify_fsm one takes almost five times longer to complete. The memory requirements for the pn_verify grows slightly faster than for the verify_fsm. For the set of benchmarks represented real hardware designs the pn_verify method is better than the verify_fsm when comparing both time and memory needed for verification.

In Table 3 the logic verification results are presented. In general, the logic verification statistics have similar characteristics to those of behavioural verification. However, performances of the pn_verify routine are considerably better than those of verify_fsm for almost all of the examples. For example, for the **smoke** set of benchmarks the verify_fsm approach takes almost ten times longer than pn_verify.

5 Conclusions

A new method for the behavioural and logic verification of parallel controllers has been developed. Experimental results for a number of parallel controllers show that significant improvements in speed and the use of memory can be obtained this way, compared with equivalent techniques which are targeted for an FSM verification, for almost all types of parallel controllers. It should be noted that proposed algorithm produces the best results for the **traffic** examples. These designs have the large number of concurrent processes (e.g. traffic4 is in 85% parallel), and so it seems that the proposed algorithm is especially suitable for such a type of controllers.

The main directions for future work in this area have become evident. Since the reachability graph generation procedure is the bottleneck of the method presented, this problem should be carefully investigated.

References

[1] K. Bilinski. *Parallel Controller Synthesis and Verification — Petri Net Based CAD Tool.* MSc thesis, University of Bristol, 1993.

[2] O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of Computer-Aided Verification 2nd International Conference CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer-Verlag, June 1990.

[3] S. Devadas, H-K.T. Ma, and A.R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Transactions on Computer Aided Design*, 7(6):713–722, June 1988.

[4] A. Ghosh, S. Devadas, and A.R. Newton. Test generation and verification for highly sequential circuits. *IEEE Transactions on Computer Aided Design*, 10(5):652–667, May 1991.

[5] A. Ghosh, S. Devadas, and A.R. Newton. *Sequential Logic Testing and Verification.* VLSI, Computer Architecture and Digital Signal Processing. Kluwer, 1992.

[6] S. H. Hwang and A.R. Newton. An efficient verifier for finite state machines. *IEEE Transactions on Computer Aided Design*, 10(3):326–334, March 1991.

[7] T. Kozlowski. *Petri Net Based CAD Tools for Parallel Controller Synthesis.* MSc thesis, University of Bristol, 1993.

[8] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):548–580, 1989.

[9] J. Pardey, T. Kozlowski, J. Saul, and M. Bolton. State Assignment Algorithms for Parallel Controller Synthesis. In *Proceedings of the IEEE International Conference on Computer Design*, pages 316–319. IEEE Computer Society Press, 1992.

[10] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Sadanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis.* University of California, Berkelay, May 1992. Memorandum No. UCB/ERL M92/41.