Synchronous Parallel Controller Synthesis from Behavioural Multiple–process VHDL Description

Krzysztof Bilinski† Jaroslaw Mirkowski‡ Erik L. Dagless†

† Dept of Electrical & Electronic Eng.
University of Bristol
Bristol BS8 1TR, United Kingdom

‡ Dept of Computer Engineering & Electronics
Higher College of Engineering
65–246 Zielona Gora, Poland

Abstract

A unified framework and associated algorithms for a behavioural synthesis of parallel controllers from a multiple–process VHDL specification is presented. An extension to FSMs, based on Petri nets, is used as an internal representation of an concurrent system during the synthesis. The VHDL simulation cycle implications are explicitly implemented into the Petri net model. This model is next decomposed into a set of well formed sub–controllers and a state assignments is generated.

1 Introduction

A performance of digital systems can be increased by applying efficient tools which explore parallelism within the design. VHDL supports process level parallelism. However, some of its simulation–oriented characteristics make VHDL difficult to apply in synthesis, namely:

- Signals are updated only at the beginning of the next simulation cycle. Thus, the postponement of signals update has to be reflected in the internal representation.
- VHDL processes execute asynchronously and are synchronized using wait statements.

As a result most of the high level synthesis systems restrict themselves to single process and variables only [2]. The system which follows the implication of the the VHDL simulation cycle in compilation is CAMAD [2], where the internal representation is realised in terms of extended timed Petri net (ETPN). The solution applied in CAMAD has a major drawback in case of signal assignment, namely the assignment is done implicitly (i.e. is not represented in the control structure of the representation) and is done at the end of the actual simulation cycle instead of at the beginning of the next one.

The primary focus in the area of controller synthesis has been in developing efficient techniques for the state assignment for finite state machines (FSM). The FSM techniques do not support any explicit representation of the multiple–process systems. An alternative approach is to use Petri nets as a formal specification of the design. Behavioural analysis of the controller can be performed, using well defined techniques from Petri net theory based on a symbolic net execution. The Petri net symbolic traversal techniques have originated from the methods which are used in sequential circuit verification [1]. The reported methods prove to be capable of dealing with large nets, due to the efficient data representation provided by BDDs.

The decomposition of the controller specification into a set of linked sub–controllers may yield several advantages [5]:

- The reduced critical path delay through the distributed controller and shorter control lines between each sub–controller and its data–path promote faster clocking.
- The smaller number of cells in each sub–controller shortens the routing lines, thus may result in a decrease of the overall area of the controller, it also assists with automatic layout.

The objectives of the work presented here can be formulated as follows:

- Formulate a methodology for compilation a multiple–process VHDL specification into a Petri net in such a way that the simulation cycle implication are fully followed and represented in the control and data flow representation of the specified system.
- Define the decomposition method for the Petri net specification of the controller. The method should comply with the following constraints: (i) the communication between sub–controllers should be minimised to improve routability of the entire design; (ii) places should be grouped so as to promote an optimum state assignment in each sub–controller.

2 Preliminaries

For a formal introduction to Petri nets refer to [4]. The use of a Petri net as a hardware description language has resulted in the introduction of some dedicated extensions to Petri net interpretation. A predicate which is a Boolean function of the controller's input signals may be attached to each transition. Thus, transition is enabled when all of its input places are marked, and its predicate, if present, is asserted. Moore outputs are associated with places, and they are asserted whenever the associated places have tokens. Mealy outputs are associated with transitions, and they are asserted whenever the associated transitions are enabled.

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

To model the systems with priority and shared resources the modeling power of Petri nets has been increased by introducing inhibitor and enabling arcs [4]. The enabling arc adds ability to test presents of tokens in a place while the inhibitor arc adds ability to test the absence of tokens in a place. Unlike ordinary arcs, no tokens are moved through an inhibitor or enabling arc when the transition fires. To model synchronous systems a new transition firing rule has to be introduced. All transitions are synchronized by a global clock, and so all enabled transitions fire simultaneously. A Petri net with above extensions is said to be a synchronous interpreted Petri net (SIPN).

3 Synthesis methodology

The design flow of the proposed synthesis system is show in Figure 1. The synthesis methodology consists of the following steps:

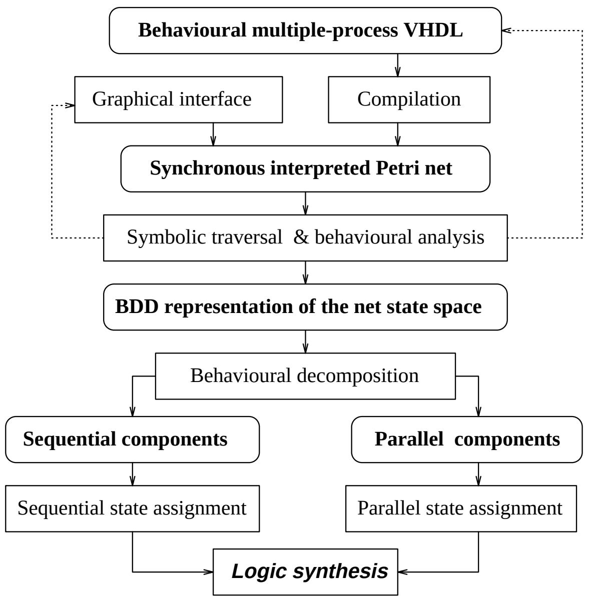
- The synthesis starts from the behavioral VHDL specification of a controller. First, this specification is translated into an SIPN. The translation method is described in detail in Section 4.
- Next, the SIPN is verified and the BDD representation of the net state–space is generated using symbolic traversal techniques [6]. Since the net analysis and traversing methods are well described in literature, they will not be discussed here.
- The symbolic representation of the net state–space is then used to behavioral decomposition of the SIPN into a set well formed sub–nets (i.e. sub–controllers). The net can be decomposed into a set of purely sequential sub–nets or sub–nets with limited concurrency. The method is presented in Section 5.
- Finally, each component is encoded separately using a unique set of flip–flops and the logic level description of the controller is produced.

4 VHDL to SIPN transformation

The compilation process of a VHDL specification into an SIPN consists of three main steps, the last two are consequences of the simulation cycle requirements.

- Transforming the operations of all processes into their respective SIPN representation. At this stage also parallelism extraction is being performed.
- Synchronisation of wait statements of all the processes.
- Correct postponement of signal assignment (signal update).

Each VHDL operation, notified as N_i, is defined as a 6-tuple: $N_i = (In_i, Out_i, Op_i, Pr_i, Su_i, Con_i)$; where: In_i is a set of input variables of N_i; Out_i is a set of output variables of N_i; Op_i is a set of operands of N_i; Pr_i is a set of predecessors of N_i; Su_i is a set of successors of N_i; Con_i is a predicate of N_i. The transformation procedure is performed as follows.



Behavioural multiple-process VHDL
Graphical interface
Compilation
Synchronous interpreted Petri net
Symbolic traversal & behavioural analysis
BDD representation of the net state space
Behavioural decomposition
Sequential components
Parallel components
Sequential state assignment
Parallel state assignment
Logic synthesis

Figure 1: Design flow.

1. *Determine all predicates;*
2. *Extract data and/or control dependencies and remove redundant dependencies;*
3. *Define type of a module for each operation, connect modules to obtain a Petri Net and reduce the net.*

Predicates, are logical conditions of executing operations. They are used to determine control dependencies, but are also reflected in the resulting SIPN. A predicate of an operation is a logical product of conditions which must be fulfilled to execute an operation.

To determine predicates the input specification is transformed in such a way, that comparisons from conditional operations are extracted and changed into operations with a logical variable on an output, for example:

before:	**after:**
if $x > 0$	$c_1 := x > 0$
then $y := 2 * k$;	*if* c_1 *then* $y := 2 * k$; *end if*;
else $y := y - 1$;	*if not* c_1 *then* $y := y - 1$; *end if*;
end if;	

An operation N_k is called dependent on N_j if N_j prepares data for N_k (data dependency) or its execution must be completed before N_k's execution can be started (control dependency), e.g. *if* N_j *then* N_k construct. List of dependencies can be easily transformed into set of predecessors and successors for each operation: if N_h is dependent on N_g, then N_g is a predecessor of N_h and N_h is a successor of N_g. Removing redundant dependencies prevents the net form unnecessary growth as every such a dependency may result in additional place and/or transition, e.g. considering the following fragment of VHDL:

```
$if\ x > 0\ then$          — 1
    $y := 2 * k;$          — 2
    $y := y - 1;$          — 3
$endif;$
```

operation 3 in this example is dependent on 1 and 2. Meanwhile, operation 2 is dependent on 1. Dependency of operation 3 on operation 1 is therefore redundant.

While an operation is represented by a place, it is additionally augmented with an input transition, starting execution of the operation. However, such modules cannot be directly connected because this would lead to a dead and/or unsafe net. Thus, additional places and/or transitions are added to the modules in order to keep the safeness and liveness of the net, which is crucial for proper functioning of designed system. Additions to modules depend on predicates of predecessors and successors of given operation. They can be divided into two groups - relating to inputs and outputs:

Inputs:
A - one predecessor;
B - several predecessors, non-conflict predicates;
C - several predecessors, conflict predicates;
D - several predecessors, mixing B and C types.

Outputs:
A - one successor,
B - several successors, conflict predicates;
C - several successors, non-conflict predicates,
D - several successors, mixing B and C types.

This forms 16 possibilities, 16 different modules of SIPN, each denoted by two letters: first relating to inputs and second - to outputs, e.g. module type AB has only one input to the enabling transition and more than one successors with conflict predicates. Predicates are said to be in conflict if they mutually exclude each other, e.g. x and $\overline{x}$. An example of modules of an operation N with a predicate C is shown in Figure 2.

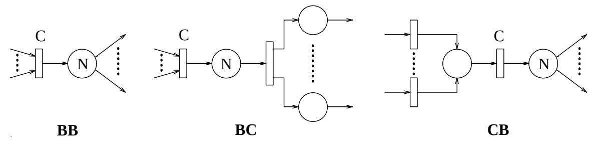


Figure 2: An example of the operating modules.

4.1 VHDL simulation cycle

Considering the definition of a simulation cycle [9] two conclusions, which are described in more details in this section, can be drawn:

- No process can be resumed until all active processes suspend.
- Signal assignment can be executed physically when respective process is resumed but before execution of any of its operations. Moreover, the process may not be suspended by a signal assignment operation.

4.1.1 Synchronisation of processes

Any VHDL process is synchronized in such a way that no process can resume its execution until all other active processes suspend. Such a synchronisation is introduced as a transition connecting all the wait statements [2].

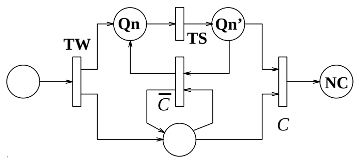


Figure 3: Synchronisation of a wait statement.

Figure 3 presents a synchronisation of a wait statement, represented by the transition **TW**, in one process. The synchronisation transition connecting all the processes is denoted as **TS**; **C** is a predicate which is true when any of the signals from the wait sensitivity list changed during recent simulation cycle. The marking of the place **NC** denotes that next simulation cycle is started and the process is resumed.

4.1.2 Signal update

A signal can be updated when two conditions are met:

- Respective operation has been selected during the execution of the process.
- The process resumed for the next simulation cycle (place **NC** from Figure 3 is marked).

The correct location of a place representing signal assignment operation is determined by a parallelism extraction step. Physical execution has to be postponed until the next simulation cycle begins. Handling both requirements can be done by substituting an original place with a new subnet as illustrated in the following example. Let OP_k be a signal assignment operation, its predicate is C_p and its direct predecessor is OP_g, the direct implementation is shown in Figure 4(a). Figure 4(b) on the other hand illustrates the implementation which results from the implications of the VHDL simulation cycle. Place **NC** is the same as in the Figure 3.

4.1.3 Resuming the process

When the process has to be resumed first the signals must be updated. It is not possible, however, to determine which of them will be updated in any particular simulation cycle as this depends on external conditions. Thus, the input transition enabling execution of the signal assignment has an enabling arc from the place denoting the beginning of next cycle. Furthermore, the marking can be removed from this place only after all necessary signals are updated. If none of the signals has been updated, the marking is removed immediately. In order to support such information a small subnet is constructed. It consists of two places and two transitions. The marking of the place **A1** denoted that no signal is being updated and the **A2** denotes updating of at least

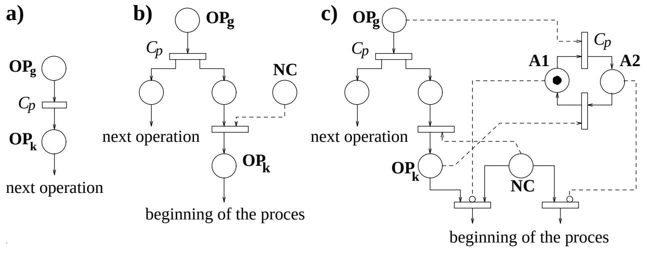


Figure 4: Signal assignment: a) without simulation cycle implementation; b) postponed according to the simulation cycle requirements; c) postponed according to the simulation requirements with correct signal updating.

one signal. Depending on this information the marking of the place **NC** is removed in one of the above mentioned ways as it is illustrated on Figure 4(c), which is an updated version of Figure 4(b).

5 Behavioural decomposition

Having an SIPN constructed and its properties verified the behavioural decomposition of the SIPN is performed. Let PN be a safe Petri net and Ω the set of all markings of PN. A set of markings of PN can be represented as an vector $M = \{m_1, m_2, ..., m_k\}$, where $m_i = (\mu_1\, \mu_2 \ldots \mu_n)$, and μ_i represents number of tokens in the place p_i, which can be 0 or 1. A set of all markings *reachable* from the initial marking M_0 is denoted $[M_0\rangle$. The characteristic function χ_M of a set of markings $M \subseteq \Omega$ is a Boolean function: $\chi_M : \Omega \rightarrow \{0, 1\}$, that evaluates to 1 on the vertices belonging to M, otherwise it is 0 [6]. The characteristic function can be efficiently represented using BDDs — Figure 5.

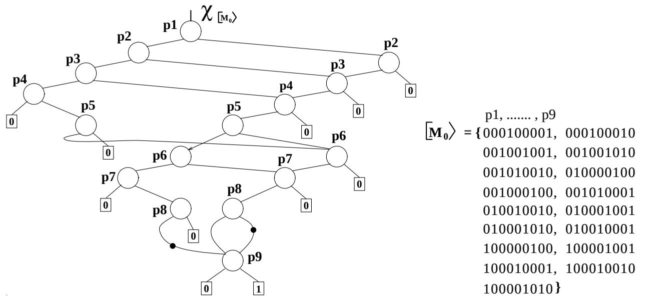


Figure 5: BDD representation of the set $[M_0\rangle$.

The Petri net decomposition algorithm operates on the characteristic function $\chi_{[M_0\rangle}$. To simplify the decomposition process, the characteristic function $\chi_{[M_0\rangle}$ is simplified prior to the decomposition.

5.1 Characteristic function simplification.

The simplification algorithm consists of the following steps. First, a pair p_i and p_j is identified, such that both p_i and p_j has the same marking relationship with all other places, i.e: *for all* $p_k \in P$:

$$\begin{cases} \chi_{[M_0\rangle} * p_i * p_k \neq 0 \Leftrightarrow \chi_{[M_0\rangle} * p_j * p_k \neq 0 \\ \chi_{[M_0\rangle} * p_i * p_k = 0 \Leftrightarrow \chi_{[M_0\rangle} * p_j * p_k = 0 \end{cases} \quad (1)$$

Next, p_i and p_j are merged by forming their Boolean product or sum and replaced by a new variable p_i:

$$p_i = \begin{cases} p_i.p_j & if\ \chi_{[M_0\rangle} * p_i * p_j = 0 \quad (a) \\ p_i + p_j & if\ \chi_{[M_0\rangle} * p_i * p_j \neq 0 \quad (b) \end{cases} \quad (2)$$

Finally, p_j is removed from the characteristic function. The algorithm is reiterated until no further simplification can be achieved. Figure 6 shows the the simplified characteristic function χ_R of the function $\chi_{[M_0\rangle}$ shown in Figure 5. For simplicity, the expressions representing Boolean product or sum will be explicitly used, instead of using the variables by which they are replaced, e.g. $p_5.p_6$ instead of p_5.

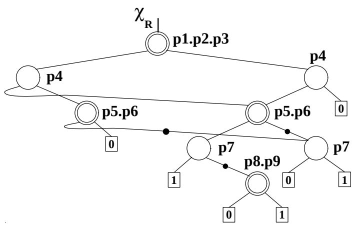


Figure 6: The reduced characteristic function.

5.2 Decomposition algorithm.

Let $G = (V, E)$ be a weighted relation graph, where: V is a set of vertices, each of which represents a variable of the χ_R; E is a set of arcs, where an arc connects two vertices if the matching variables of χ_R comply with the following criterion (i.e. two places are sequential related to each other):

$$\chi_R * p_i * p_j = 0 \quad (3)$$

To each vertex a weight is assigned. If a variable of χ_R represents a product or sum of other variables then the weight equals the number of flip–flops needed to implement the variable, otherwise the weight is 0. An example of the weighted relation graph is shown in Figure 7(a).

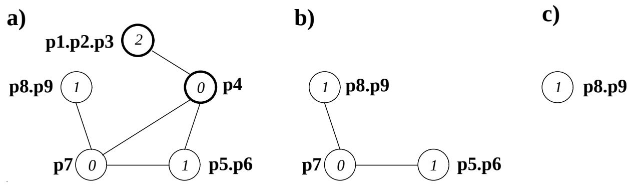


Figure 7: Weighted relation graphs.

A clique of a graph G is a subset of vertices $C \subseteq V$ such that if vertices $c_i, c_j \in C$ then there is an arc $e_k \in E$ which connect these two vertices. Each clique in the graph defines one component (i.e. sub–net) of the SIPN. The problem of finding a maximum clique is known to be a NP–complete [7]. Here, a heuristic algorithm, which works in polynomial time to find out the first good irredundant solution, with respect to the decomposition constraints is presented:

1. Identify a vertex v_i, such that for all $v_k \in V$:
$$wg(v_k) + |v_k| \leq wg(v_i) + |v_i| \qquad (4)$$
where: $wg(v)$ returns the vetrex's weight and $|v|$ is the number of arcs connected to the vertex.
2. Add the identified vertex to the clique C_l.
3. Identify another vertex v_j, which is not in the C_l, satisfies the criterion (4) and the following condition:
$$\underset{v_i \in C_l}{\forall} \; \underset{e \in E}{\exists} \; v_i \overset{e}{-} v_j \qquad (5)$$
if the vertex v_j cannot be identified then the construction of the clique C_l is completed, otherwise go to the Step 2.

   Applying these steps to the graph in Figure 7(a), the vertex $p_1.p_2.p_3$ is selected first. Next the vertex p_4 is also identified in Step 3. The first identified clique is denoted using thick circles in Figure 7(a).
4. Remove from the graph all vertices constituting the clique C_l.
5. Repeat the above steps until all vertices have been removed from the graph.

Employing the above algorithm to the graph shown in Figure 7(a), the three cliques have been selected: $C_1 = \{p_1.p_2.p_3,\ p_4\}$, $C_2 = \{p_5.p_6,\ p_7\}$, $C_3 = \{p_8.p_9\}$. Elements of each clique are sequentially related to each other. A clique (component) is said to be complete if its elements comply with the following criterion:

$$\chi_R * \sum_{v_i \in C} bdd\_variable(v_i) = \chi_R \qquad (6)$$

where: $bdd\_variable(v)$ returns the bdd variable corresponding to the vertex v. If elements of any clique do not comply with criterion (6), an extra vertex must be added to the clique, which represents an idle state of a component (i.e. a state in which no places have a token).

In the presented example, the elements of C_2 and C_3 do not comply with criterion (6), thus extra elements have been added to these cliques. The three components are defined as follows:

$$C_1 = \{p_1.p_2.p_3,\ p_4\} \quad C_2 = \{p_5.p_6,\ p_7,\ p_{idle_1}\}$$
$$C_3 = \{p_8.p_9,\ p_{idle_2}\}$$

The p_{idle} places do not effect the operation of the controller, since they do not have outputs attached to them, but they may increase the number of flip–flops required during a state assignment.

The algorithmic complexity of the graph generation procedure is $O(n^2)$, while clique construction is done in $O(n^2 + 2 * n)$ steps, where n is the number of variables of characteristic function χ_R (i.e. the number of places of the reduced SIPN).

5.3 State assignment

State assignment assigns a state code to each place in the decomposed SIPN. Every component is encoded using a separate set of flip–flops. Two encoding approaches have been implemented:

- If the SIPN is decomposed into a set of sequential components, i.e. only criterion (2.a) was applied then sequential state assignment techniques such as one–hot–code, Gray or binary encoding are used.
- If the SIPN is decomposed into a set of well formed concurrent components, i.e. both (2.a) and (2.b) were applied. Then the places in each components can share state variables to reduce the number of flip–flops in the state register, subject to the following constraint: the places which hold tokens simultaneously must have non–orthogonal codes. Two codes are said to be non–orthogonal if they differ by at least one state variable.

The final state assignment for the current example is shown in Table 1.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
Q_1	1	1	0	0					
Q_2	1	0	1	0					
Q_3					1	1	0		
Q_4					1	0	1		
Q_5								1	1
Q_6								1	0

Table 1: Final state assignment.

6 Experimental results

All presented algorithms have been implemented in C and a CAD system for high–level synthesis of parallel controller was created. In this section, the synthesis results are described. All benchmarks were run on a Sun SPARC–Station2 computer with 64 MByte of memory.

In Table 2 the translation results obtained using the presented approach (*SIPN*) and those produced by the use of the CAMAD system are shown. The cost function (*cf*) of the parallelism extraction is defined as follows:

$$\underset{p_i p_j \in supp(\chi_{[M_0\rangle})}{\forall} \chi_{[M_0\rangle} * p_i * p_j \neq 0 \;\Rightarrow\; par = par + 1;$$
$$cf = (|supp(\chi_{[M_0\rangle})|^2 - par)/(|supp(\chi_{[M_0\rangle})|^2) * 100;$$

where: $supp(\chi_{[M_0\rangle})$ is a support of $\chi_{[M_0\rangle}$.

The CAMAD system produces a slightly smaller nets due to the lack of the correct signal assignment procedure, e.i. signal assignment is done at the end of the

	VHDL			SIPN			CAMAD		
name	ln	pr	wt	pl	tr	cf	pl	tr	cf
armstr	139	5	5	95	102	69	79	88	49
move_mc	148	3	3	53	49	59	38	45	40
rs232	166	2	2	44	45	42	38	46	31
mark1	71	1	0	13	20	0	18	24	0

ln: number of lines; lr: number of processes;
wt: number of waits; pl/tr: places/transitions;

Table 2: Translation results.

							BDD decomposition				Place encoding			
							#ff	area	delay	T_{cpu}	#ff	area	delay	T_{cpu}
name	in	out	pl	tr	$\|[M_0\rangle\|$	$\chi_{[M_0\rangle}$	–	$[\mu m^2]$	[ns]	[s]	–	$[\mu m^2]$	[ns]	[s]
frcntr	15	21	62	64	21575805	8086	37	565001	7.82	9720	-	-	-	-†
minctr	13	17	46	48	93242	2006	24	395930	7.22	658	-	-	-	-†
armstr	12	1	91	102	54933	2603	46	487864	6.59	432	47	541360	9.23	5012
sm7	6	18	56	42	28516	879	38	478949	5.00	147	38	487784	6.13	2627
bar	9	12	30	31	18797	308	18	282246	4.29	135	18	283651	4.88	1351
rascas	8	4	28	28	1214	487	16	332678	4.10	171	19	379220	3.80	240
gf2	10	4	20	20	485	106	12	173036	4.39	65	12	157783	4.61	95
rs232	10	24	44	45	165	248	24	188699	7.06	132	23	215711	11.95	203
am2109	34	90	130	153	130	257	8	158791	6.28	102	8	166621	6.31	100
spool	4	19	28	24	77	64	16	250601	5.86	95	19	356621	7.57	91

†The computation cannot be completed within 24 hour time limit.

in/out: number of inputs/outputs, ff: number of flip–flops, area: area of the final implementation, delay: max. critical path delay, T_{cpu}: synthesis cpu run–time.

Table 3: Synthesis results.

present simulation cycle instead of at the beginning of the next one. Thus, different result are produced by VHDL simulator and CAMAD simulator. Further, the SIPNs are significantly more parallel then their CAMAD counterparts.

To evaluate the efficiency of decomposition method the synthesis results were compared with an alternative parallel implementation based on a hierarchical place encoding approach presented in [3]. Each of the examples, after the state assignment, was optimized using the Berkeley synthesis system SIS ver.1.2 [8] and synthesized using the Cadence Framework II design system with the ES2 ECPD10 $1\mu m$ cell libraries.

Table 3 shows the synthesis results. Summarizing the experimental results we can observe, that for all the examples the decomposition implementations produce better results when compared to their encoding counterparts — mainly because of the better grouping of flip–flops during the state assignment and shorter routing lines. The presented method is also able to handle the synthesis of much more complex controllers than the alternative approaches.

7 Conclusion

A novel approach to high–level synthesis of parallel controllers from VHDL specifications has been presented. Its novelty lies in direct implementation of all implications of the VHDL simulation cycle into the internal Petri net representation of the controller. This net is next decomposed into a set set of well formed sub–controllers using a method based on symbolic manipulation of the controller state–space. The application of the proposed methodology ensures a one-to-one correspondence between the results of a VHDL simulator (applied to the input specification) and the results of the synthesis. The experimental results clearly demonstrate the advantages of the method over the alternative methods.

References

[1] 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.

[2] P. Eles, K. Kuchcinski, Z. Peng, and M. Minea. Synthesis of VHDL Concurrent Processes. In *Proceedings of the European Design Automation Conference EURO–DAC'94*, pages 540 – 545, Grenoble, September 19–23, 1994.

[3] T. Kozlowski, E. Dagless, J. Saul, M. Adamski, and J. Szajna. Parallel Controller Synthesis using Petri Nets. *IEE Proceedings – Computers and Digital Techniques*, 142(4):263 – 271, July, 1995.

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

[5] J. Pardey and M Bolton. Logic Synthesis of Synchronous Parallel Controlers. In *Proceedings of the IEEE International Conference on Computer Design*, pages 454–457. IEEE Computer Society Press, 1991.

[6] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri Net Analysis Using Boolean Manipulation. In R. Valette, editor, *Proceedings of 15th International Conference: Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 416 – 435. Springer-Verlag, June 1994.

[7] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms, Theory and Practice.* Prentice Hall, 1977.

[8] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, 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.

[9] *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, 1988.