Logic Synthesis of Handshake Components using Structural Clustering Techniques

Francisco Fernández-Nogueria[*]
Universitat Politècnica de Catalunya
Barcelona, Spain

Josep Carmona
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

A methodology to optimize handshake circuits is presented. The approach selects clusters of the initial handshake network for which signals representing internal channels within a cluster are hidden. To guarantee asynchronous implementability on the resulting cluster, state encoding is applied using modern structural techniques. The theory of Petri nets is used to identify clusters for which the structural techniques perform successfully. Finally logic synthesis is employed for each reencoded cluster. The approach is integrated into the Balsa synthesis flow and may represent a significant improvement with respect to the local optimizations typically applied. Experimental results in area and performance have been obtained to measure the optimization on typical Balsa examples.

1. Introduction

Asynchronous circuits represent a robust alternative for overcoming the problems of current and future technologies [14]. The nightmares of the synchronous paradigm like power dissipation, clock distribution, EMI, worst case performance among others are naturally avoided when one gets rid of the clock [3].

However, asynchronous circuits appear seldom in current technologies. The reason for this is simple: a circuit that lacks a global coordinator is difficult to design and verify. In the last decades, theories, methodologies and tools for the design and verification of asynchronous circuits have appeared, but their scope have been mostly academic. Moreover, these asynchronous paradigms traditionally have used as specification language formal models like automata or Petri Nets [20, 13, 10], which are not well-suited as front-end for the design of large and complex systems.

Hardware Description Languages (HDL) offer a simple way to design circuits. Many nuisances of the design process are hidden or automated, and allow the designer to have a system-level view of the circuit. The complexities of asynchronous circuit design can also be hidden by using an HDL as a front-end. With this idea in mind, the asynchronous community has provided some HDLs for the asynchronous design [4, 11]. Typically those programming environments transform the program, using a syntax-directed translation of each primitive, into a netlist of handshake components. Latterly each handshake component can be synthesized separately into an asynchronous circuit. Hence the size of the resulting circuit is linear with respect to the size of the HDL program. This can limit the use of current asynchronous HDLs when area and/or performance is a key factor.

Logic synthesis achieves global optimizations that can improve in orders of magnitude the local (*peephole*) optimizations applied in asynchronous HDLs [9, 6]. In [8], a back-end to incorporate logic synthesis into the Balsa system was presented. The work showed the tangible improvements that can be obtained by optimizing the netlists of handshake circuits.

In this paper we provide a Petri net-based back-end to the Balsa system, offering resynthesis capabilities that include state encoding and logic synthesis of selected clusters of handshake components. The approach can be considered a follow-up of previous work [15, 17, 5, 8, 19], with the differences listed below:

1. State-based methods are used in [17, 5, 8], thus suffering from the state space explosion problem. Hence their application is limited to small specifications. In the work presented in this paper, modern structural methods for state encoding and synthesis [6, 7] are employed, allowing large specifications to be handled.

2. Petri nets are used as intermidiate language, whereas the underlying formalism in [8] for synthesis are burst-mode machines, that impose limitations on modeling

[*]Supported by the Universitat Politècnica de Catalunya (predoctoral scholarship UPC for investigation).

the inherent concurrency of asynchronous systems.

3. A structural clustering approach guides the composition of handshake components, which are described by labeled Petri nets, into clusters. Those clusters grow as far as the induced Petri net composition of the selected components belongs to a class for which structural methods perform well. A *blind* clustering is used in the Petri net-based approaches [15, 17, 5], often deriving unrestricted clusters that synthesis methods can not handle.

4. No change in the specification language is required: the designer might benefit from the optimizations provided in this paper without even knowing that they are applied. This differs from the approach in [19], where a data-oriented Balsa language is presented to improve the performance of Balsa specifications. The approach presented in this paper is integrated in the Balsa synthesis flow: resynthesized clusters are translated back to Balsa implementation format.

The organization of the paper is the following: the basic knowledge required for this paper is presented in Section 2. Section 3 describes the overall design flow for synthesis and optimization of Balsa systems. The structural clustering algorithm is described in Section 4 and Section 5 describes the clustering strategy for a Balsa example. Finally, a set of experiments on typical Balsa examples is studied in Section 6.

2. Basic Theory

A Petri Net (PN) is a 4-tuple, $N = (P, T, F, m_0)$, where P is a finite set of *places*, T is a finite set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* and $m_0 \in \mathbb{N}^{|P|}$ is the initial *marking*. Given a node $x \in P \cup T$, the set ${}^\bullet x = \{y|(y, x) \in F\}$ is the *preset* of x and the set $x^\bullet = \{y|(x, y) \in F\}$ is the *postset* of x.

A marking assigns to each place a nonnegative integer. If a marking assigns to place p a nonnegative integer k, we say that p is marked with k tokens and we place k dots in place p. A marking, denoted by m, is a $|P|$-vector where the pth component, denoted by $m(p)$, is the number of tokens in place p.

A marking in a PN is changed according to the following firing rule: a transition t is enabled if each input place p of t is marked; an enabled transition may or may not fire; a firing of an enabled transition t removes one token from each input place p of t, and adds one token to each output place p of t.

A marking m_n is reachable from a marking m_0 if there is a sequence of firings $\sigma = t_1 t_2 \ldots t_n$ that transforms m_0 to m_n, denoted by $m_0[\sigma\rangle m_n$. The set of all possible markings reachable from m_0 is denoted by $[m_0\rangle$. The *reachability graph* can be obtained considering the set of reachable markings as the set of states and the transitions among these markings as the transitions between the states.

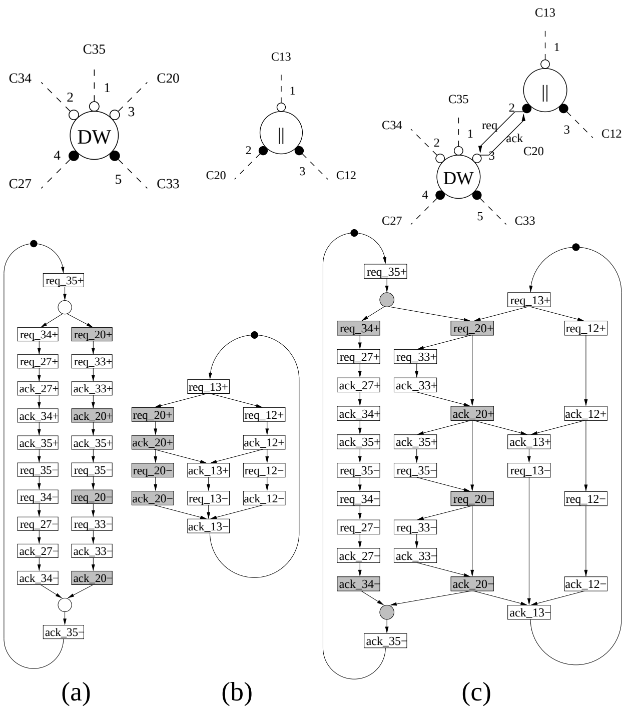


Figure 1. (Top) HC components and their connection, (Bottom) STGs and their parallel composition.

Four special PN classes [16] are of interest in this paper. A PN N is a:

- *Marked graph* (MG) if $\forall p \in P : |{}^\bullet p| = |p^\bullet| = 1$.
- *State machine* (SM) if $\forall t \in T : |{}^\bullet t| = |t^\bullet| = 1$.
- *Free-choice* (FC) if $\forall p_1, p_2 \in P : p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow |p_1^\bullet| = |p_2^\bullet| = 1$.
- *Asymmetric Choice* (AC) if $\forall p_1, p_2 \in P : p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow p_1^\bullet \subseteq p_2^\bullet$ or $p_1^\bullet \supseteq p_2^\bullet$.

Clearly, considering set inclusion as class inclusion, the following holds: MG, SM $\subset$ FC $\subset$ AC.

Given places p_1 and p_2, where $p_1^\bullet \cap p_2^\bullet \neq \emptyset$, FC is violated if $|p_1^\bullet| \neq 1$ or $|p_2^\bullet| \neq 1$ and AC is violated if $p_1^\bullet \not\subseteq p_2^\bullet$ and $p_1^\bullet \not\supseteq p_2^\bullet$. We will call these violations *FC-violation* and *AC-violation*. If the class is a parameter C, we will simple call *C-violation*.

2.1. Signal Transition Graphs

To model digital circuits, the events of a PN can be interpreted as signal changes. A *Signal Transition Graph* (STG) is a triple $G = (N, \Sigma, \Lambda)$, where $N = (P, T, F, m_0)$ is a PN, Σ is a set of signals, partitioned into input, internal and output signals, and $\Lambda : T \to \Sigma \times \{+, -\} \cup \{\epsilon\}$ is the labeling function which maps rising and falling signal transitions to transitions in the PN. The symbol ϵ can be assigned to any transition to denote a silent event in the system.

Definition 1 (Parallel Composition) *Given STGs G_1 and G_2, their parallel composition is denoted by $G_1||G_2 = ((P_{||}, T_{||}, F_{||}, m_{0||}), \Sigma_{||}, \Lambda_{||})$, where:*

$$
\begin{aligned}
P_{||} &= P_1 \times \{*\} \cup \{*\} \times P_2 \\
T_{||} &= \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, \\
&\qquad \Lambda_1(t_1) = \Lambda_2(t_2) \in \Sigma_1 \cap \Sigma_2 \times \{+, -\}\} \\
&\quad \cup \{(t_1, *) \mid t_1 \in T_1, \Lambda_1(t_1) \notin \Sigma_1 \cap \Sigma_2 \times \{+, -\}\} \\
&\quad \cup \{(*, t_2) \mid t_2 \in T_2, \Lambda_2(t_2) \notin \Sigma_1 \cap \Sigma_2 \times \{+, -\}\} \\
F_{||} &= \{((p_1, p_2), (t_1, t_2)) \mid (p_1, t_1) \in F_1 \ or \ (p_2, t_2) \in F_2\} \\
&\quad \cup \{((t_1, t_2), (p_1, p_2)) \mid (t_1, p_1) \in F_1 \ or \ (t_2, p_2) \in F_2\}
\end{aligned}
$$

$$
m_{0||}((p_1, p_2)) = \begin{cases} m_{01}(p_1) \ if \ p_1 \in P_1 \\ m_{02}(p_2) \ if \ p_2 \in P_2 \end{cases}
$$

$$
\Sigma_{||} = \Sigma_1 \cup \Sigma_2
$$

$$
\Lambda_{||}((t_1, t_2)) = \begin{cases} \Lambda_1(t_1) \ if \ t_1 \in T_1 \\ \Lambda_2(t_2) \ if \ t_2 \in T_2 \end{cases}
$$

Informally, the STG representing the parallel composition represents the joint behavior of the participating STGs. Bottom of figure 1(c) shows the parallel composition of the STGs in bottom of figures 1(a)-(b). The shared events are depicted in transitions with grey background.

2.2. STG modeling of Handshake Control Circuits

Handshake Circuits are asynchronous circuits composed of handshake components (HC) and channels. They are obtained by a syntax-directed translation from a CSP-like language like Tangram [4] or Balsa [11]. The handshake components communicate through channels using a handshake protocol. This protocol can be described with an STG. In [12], STG characterizations of the more representative control HCs in Balsa were presented, based on the formal definition from [2][1]. An example of such characterization can be found in Figure 1(a)-(b): the *DecisionWait* and *Concur* HCs and the STGs modeling their behavior are depicted. Along the paper, we will use the notation $STG(x)$ to refer to the STG describing the protocol behavior of the HC x.

The connection of a pair of HCs x and y through a shared channel will be denoted $Conn(x, y)$. The protocol of this connection can also be described as an STG, and corresponds to the parallel composition of the STGs for x and y: $STG(Conn(x, y)) = STG(x) \,||\, STG(y)$. Using iteratively the composition operator, one can build an STG representing a cluster of HCs from a given handshake circuit. Figure 1(c) shows a possible connection of the $DecisionWait$ and $Concur$ HCs and the STG modelling their connection behavior.

The class of asynchronous circuits that we focus in this paper is Speed-Independent (SI) circuits, which operate correctly regardless of the delays on their gates. The conditions for a specification to be correctly implemented under the SI model are [10]: boundedness, consistency, complete state coding and persistency.

[1]Some of the components from [12] have been adapted to the improved versions as described in [18].

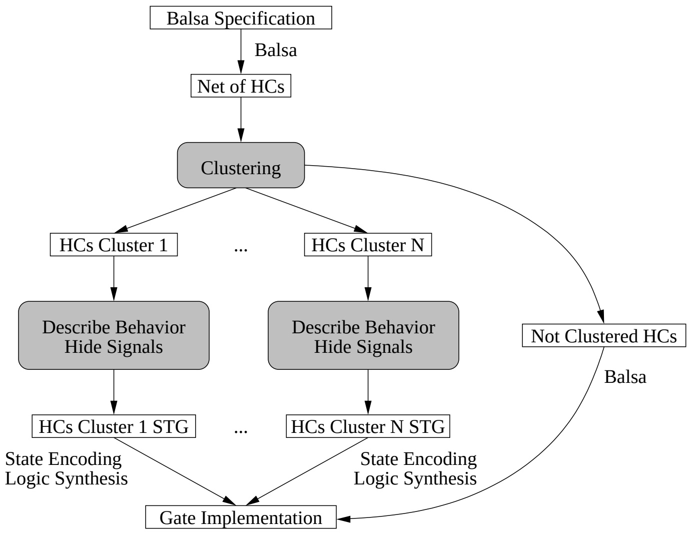


Figure 2. New Design Flow

3. Logic Synthesis of Handshake Components

Given a specification in Balsa, the goal of this work is to apply state reencoding and logic synthesis to (part of) it in order to achieve global optimizations that can improve significantly the quality of the resulting SI circuit. This optimizations can not be attained when the syntax-directed translation approach is applied to the initial specification.

Informally the approach proceeds as follows (see Figure 2): starting from the net of HCs derived from the Balsa program, it iteratively selects clusters of components following a criteria. For a given cluster selected, it creates the corresponding STG, then it hides all the signals corresponding to internal channels and state signals. If internal signals are hidden, the resulting STG may have encoding conflicts that must be resolved before of applying logic synthesis. Then state encoding and logic synthesis is applied to this STG. For state encoding and synthesis, structural methods [7, 6] are used. HCs not included in a cluster (data HCs and control HCs not assigned to any cluster), are synthesized by the Balsa synthesis flow.

The use of structural methods for the synthesis enables the selection of large clusters (i.e. large STGs) that will not be synthesized if state-based methods were used instead, due to the state-space explosion problem. The possibility of applying state reencoding and logic synthesis to large clusters of HCs induces aggressive optimizations in the resulting circuits, as has been demonstrated in [8, 6]. However, provided that structural methods in [7, 6] work with an approximation of the state space of the system, they can only guarantee a solution when the STG is well-structured. The main theoretical contribution of this work is to describe how to select clusters in order to derive STGs belonging to PN classes for which structural methods will succeed, and provide a methodology to automate this selection. The following section addresses in detail these issues.

4. Structural Clustering Algorithm

The problem addressed in this section is: given a net of HCs, and a PN class C, how to derive a (maximal) set of clusters satisfying that the STG corresponding to each cluster belongs to C? This section presents a greedy algorithm for this problem.

As explained in Section 2.2, the clusters are obtained by connecting HCs that share a channel, and the behavior of their connection is described by the parallel composition of the individual STGs. The class of the STG induced by the connection depends on the selected components and the channel, and it is not necessarily the maximal of the two initial STG. Let us use the simple example of Figure 1 to illustrate this: it shows a connection between a $DecisionWait$ and a $Concur$. The former (later) is described by the STG from Figure 1(a) ((b)), with its underlying PN belonging to the SM (MG) class. Therefore both HCs can be described with the simplest classes (see the end of Section 2.1). However, their parallel composition (shown in Figure 1(c)), "jumps" to the AC class.

In the following subsections, G_i will denote an STG and C_i a HC.

4.1. PN Class of the Parallel Composition

In order to know the class of the parallel composition of two STGs, only a special part of the parallel composition must be observed:

Definition 2 (Synchronization Area) *Given STGs G_1 and G_2, their synchronization area is denoted by $Synch(G_1, G_2) = ((P_S, T_S, F_S, m_{0S}), \Sigma_S, \Lambda_S)$, where:*

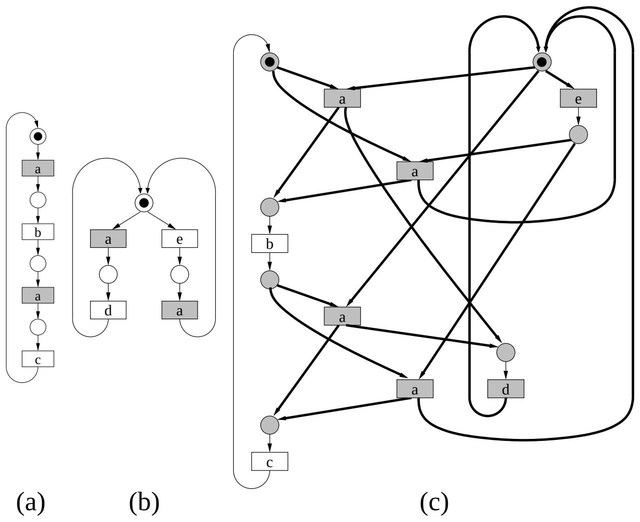


Figure 3. (a) G_1**, (b)** G_2**, (c)** $G_1||G_2$

$$
\begin{aligned}
&P_S = P_S^{pre} \cup P_S^{post}\\
&P_S^{pre} = \{(p_1,p_2) \in P_{||} \mid \exists (t_1,t_2) \in {}^\bullet(p_1,p_2),\\
&\qquad\qquad \Lambda_{||}((t_1,t_2)) \in \Sigma_1 \cap \Sigma_2 \times \{+,-\}\}\\
&P_S^{post} = \{(p_1,p_2) \in P_{||} \mid \exists (t_1,t_2) \in (p_1,p_2)^\bullet,\\
&\qquad\qquad \Lambda_{||}((t_1,t_2)) \in \Sigma_1 \cap \Sigma_2 \times \{+,-\}\}\\
&T_S = \{(t_1,t_2) \in T_{||} \mid \exists (p_1,p_2) \in {}^\bullet(t_1,t_2), (p_1,p_2) \in P_S^{post}\\
&\qquad\qquad \textit{or } \exists (p_1,p_2) \in (t_1,t_2)^\bullet, (p_1,p_2) \in P_S^{pre}\}\\
&F_S = \{((p_1,p_2),(t_1,t_2)) \in F_{||} \mid (p_1,p_2) \in P_S^{post}, (t_1,t_2) \in T_S\}\\
&\qquad \cup \{((t_1,t_2),(p_1,p_2)) \in F_{||} \mid (p_1,p_2) \in P_S^{pre}, (t_1,t_2) \in T_S\}\\
&m_{0S}((p_1,p_2)) = m_{0||}((p_1,p_2))\\
&\Sigma_S = \{\sigma \in \Sigma_{||} \mid \exists (t_1,t_2) \in T_S, \Lambda_{||}((t_1,t_2)) \in \sigma \times \{+,-\}\}\\
&\Lambda_S((t_1,t_2)) = \Lambda_{||}((t_1,t_2))
\end{aligned}
$$

Intuitively, the synchronization area contains the flow relation between places with shared transitions in their presets (postsets) and these presets (postsets). This flow relation is focused in places with more than one outgoing arc (*choice* places), which are the main responsibles for inclusion in one of the PN classes described in Section 2.1: if a place p in $G_1||G_2$ has a shared transition in its pre-set (post-set), then p and ${}^\bullet p$ ($p^\bullet$) will be in $Synch(G_1, G_2)$. The synchronization area of the STGs depicted in Figure 3(a)-(b) is described with grey background on the parallel composition in Figure 3(c). Note that the output arc of the not shared transition d is in the synchronization area, but its input arc is not there. It is due to the existence of a shared transition a in the preset of the output place of d, and the inexistence of a shared transition in the postset of its input place.

When the PN classes of G_1, G_2 and $Synch(G_1, G_2)$ are known, the PN class of $G_1||G_2$ can be obtained using the following propositions:

Proposition 1 *Inclusion of a parallel composition in a PN class:*

$$\begin{array}{c} C \in \{FC, AC\} \\ G_1, G_2, Synch(G_1, G_2) \in C \quad \Rightarrow \quad G_1 || G_2 \in C \end{array}$$

Proof: See appendix A. □

Proposition 2 *Exclusion of a parallel composition in a PN class:*

$$\begin{array}{c} C \in \{FC, AC\} \\ (G_1 \notin C \text{ or } G_2 \notin C \\ \text{or } Synch(G_1, G_2) \notin C) \quad \Rightarrow \quad G_1 || G_2 \notin C \end{array}$$

Proof: See appendix A. □

For instance, using the example of Figure 1, the synchronization area shown in solid lines in (c) contains a pair of places that violate the FC condition, but satisfy the AC principle. Applying Proposition 1 with $C = \text{AC}$, the STG in (c) is at most in the AC class. Applying Proposition 2 with $C = \text{FC}$, the STG in (c) is not in the FC class, and therefore it is an AC PN.

4.2. PN Class of the Synchronization Area

Propositions 1 and 2 point to the PN class of the synchronization area as the main element to look when the class of the parallel composition must be found. As it has been suggested in the example of the previous section, it is only needed to look at the choice places for realizing the PN class of the synchronization area. A choice in $Synch(G_1, G_2)$ is either originated from a choice in G_1 or G_2, or it arises in the parallel composition by a sharing of a transition. Hence the PN class of $Synch(G_1, G_2)$ depending on the origin of its choices has been studied.

Table 1 summarizes, for several situations of shared transitions and their presets in G_1 and G_2, the resulting structure and its the corresponding PN class in $Synch(G_1, G_2)$. The first two columns show if G_1 (G_2) has a choice in the preset of the shared transitions and/or whether it has more than one copy of the shared transition. Column $Synch(G_1, G_2)$ shows the corresponding structure in the synchronization area, and the PN class for this portion. The table shows typical situations of sharing when the STGs represent handshake components[2].

For example, the second row characterizes the following situation: G_2 has a choice in the preset of the shared transition a, G_1 does not have a choice in the preset of a, and G_1 and G_2 have only one copy of a. Then the parallel composition of G_1 and G_2 will contain the PN structure shown in the third column, which belongs the AC class, and it does not belong to the FC class.

It is important to realize that using Table 1, one may infer the PN class of the synchronization area without actually building the parallel composition. This can be done by looking individually to all the transition sharing situations and obtain the more general class that includes all them.

[2]PN stands for the class of general Petri nets.

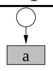
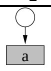
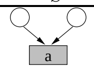
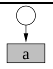
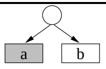
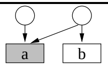
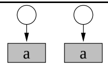
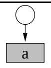
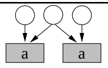
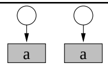
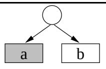
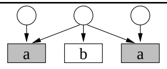
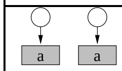
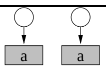
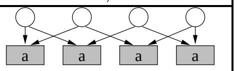
G_1	G_2	G_S
		 FC
		 $AC, \overline{FC}$
		 $AC, \overline{FC}$
		 $AC, \overline{FC}$
		 $PN, \overline{AC}$

Table 1. PN class of the Synchronization Area

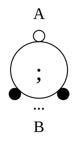
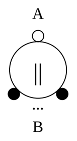
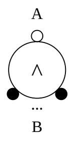
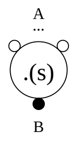
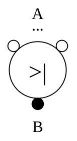
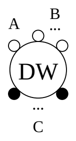
Sequence Optimised	Concur	Fork	Synch	Call	Decision Wait
					

Table 2. N ports HCs

4.3. PN Class of HCs Connection

Table 1, together with the knowledge of the PN classes of $STG(C_1)$ and $STG(C_2)$ are enough for determining a priori the class of $STG(Conn(C_1, C_2))$. Depending on the ports connecting the two HCs, different outcomes can arise. The port structure, for some Balsa HCs is shown in Table 2. For instance the $Concur$ HC has one passive port (A) and several active ports (B).

Table 3 enumerates the PN class that arises when connecting some Balsa HCs on particular ports. In each case, the HC is described together with a partition on its ports. Cells filled with $-$ denote forbidden connections. For instance, looking at the row for the Synch HC, it states that a Synch can be connected through its passive ports to the active ports of a $DecisionWait$ and the resulting connection induces a FC PN. However, if the connection is done instead through the active ports of the Synch and the passive ports of the $DecisionWait$, the resulting connection induces an AC PN.

		DecisionWait			Call		Synch		Sequence Concur, Fork	
		C	B	A	B	A	B	A	B	A
Sequence Concur, Fork	A	FC	–	–	$AC,\overline{FC}$	–	FC	–	FC	–
	B	–	$AC,\overline{FC}$	$AC,\overline{FC}$	–	$AC,\overline{FC}$	–	FC	–	
Synch	A	FC	–	–	$AC,\overline{FC}$	–	FC	–		
	B	–	$AC,\overline{FC}$	$AC,\overline{FC}$	–	$AC,\overline{FC}$	–			
Call	A	$AC,\overline{FC}$	–	–	$AC,\overline{FC}$	–				
	B	–	$AC,\overline{FC}$	$PN,\overline{AC}$	–					
DecisionWait	A	$AC,\overline{FC}$	–	–						
	B	$AC,\overline{FC}$	–							
	C	–								

Table 3. PN Subclass of HCs Connection

Let us go back to the example of Figure 1 to illustrate how Table 3 has been filled, by applying the knowledge in Table 1. In the figure, a connection between a 3-ports $Concur$ and a 5-ports $DecisionWait$ is considered. Looking at the shared events (events corresponding to the signals req_20 and ack_20), all them fall into the two following situations: 1) events req_20-, ack_20+ and ack_20- correspond to the situation described in the first row of Table 1 and therefore induce a FC PN, and 2) event req_20+ corresponds to the situation described in the second row of Table 1, hence inducing a AC PN. Taking the more general class of the two situations, the cell in Table 3 for the combination considered contains $AC, \overline{FC}$.

4.4. Iterative Clustering Algorithm

In this section we describe an algorithm to iteratively grow clusters of HCs under structural conditions. Each cluster obtained is guaranteed to be in a certain PN class. To bound the class of the STG corresponding to each cluster is crucial for the use of structural methods, given the limitations of such approaches regarding the structure of the nets. Informally, the algorithm searches for HCs that can be clustered, using function *cluster* and replaces the set of HCs assigned to a cluster by the new HC created that represents the whole cluster which has been optimized. This process is iterated until no more cluster can be created. Let us describe informally the main functions involved in the clustering algorithm.

Function *cluster* searches control HCs in the HC graph that can be initially considered for growing a cluster. Only control HCs with corresponding STG within the PN class *C* are considered. When a control HC satisfying the requirements is found, the recursive function *expand* is invoked to grow the cluster from the HC component.

```
function cluster ( g : graph ; C : integer )
return cl : set of vertexs
  for each v ∈ vertexs(g)
    if is_control(v)
    and PN_class(v) ≤ C then
      add(cl, v)
      expand(cl, C)
      return cl
  return empty_set()
```

Function *expand* adds HCs to a possible cluster (*cl*) preserving the PN class *C* of its STG. It searches for a control HC (*n*) which is connected to *cl* and whose STG belongs to *C*. If it finds one, it also checks if $Synch(STG(cl), STG(n))$ belongs to *C*. To do so, it uses function *synch_area_PN_class*. If the function returns true, the preservation of *C* is ensured due to Propositions 1 and 2.

```
function expand ( cl : set of vertexs ; C : integer )
  for each v ∈ cl
    for each n ∈ neighboors(v)
      if n ∉ cl
      and is_control(n)
      and PN_class(n) ≤ C
      and synch_area_PN_class(cl, n) ≤ C then
        add(cl, n)
        expand(cl, C)
        return
```

Function *synch_area_PN_class* returns the PN class *C* of $Synch(STG(cl), STG(n))$, where *cl* is a possible cluster and *n* is a HC. The function obtains, for each HC (*nn*) connected to *n* and in *cl*, the PN class of $Synch(STG(n), STG(nn))$ using Table 3.

```
function synch_area_PN_class
( cl : set of vertexs ; n : vertex )
return C : integer
  C := FC
  for each nn ∈ neighboors(n)
    if nn ∈ cl then
```

```
        C := max(C, connection_PN_class(n, nn))
    return C
```

In general, the clusters obtained might not be maximal with respect to HC count, but we found that this greedy strategy is fast and effective into finding a good solution in practice. The approach guides the selection of HCs to be added to a cluster depending on its restrictions imposed by Table 3, i.e. the less restrictive HCs are selected first and afterwards the remaining HCs are added if possible. The following section illustrates how the algorithm applies the clustering strategy in a real example.

5. A complete example

A Balsa example is optimized to illustrate how the structural clustering algorithm works. This example has been selected due to the contrast between its FC and AC clustering. Moreover, its recursive specification allows us to analyze the implementation improvement depending on its size.

5.1. Example clustering

The specification of a $Stack$ with capacity 3 is translated into HC net using Balsa. Figure 5 shows this net, where HCs clustered by the algorithm have grey background (light or dark grey, depending on the class used for the clustering). In FC mode, HCs with light grey background are added into cluster, while the dark grey HCs will belong to another clusters. However, in AC mode, HCs in both light and dark grey are inserted into only one cluster.

Let us explain the clustering for the FC mode. HCs without FC violations on its connections are $Sequence$s, $Concur$s and $Synch$s (see table 3). Numbers in the nodes of Figure 5 represent a possible order in which nodes are visited in FC mode, assuming that the $Sequence$ HC labeled 1 is the first node added to the cluster. The exploration is done by expanding the cluster with nodes that impose less restrictions. For instance, the second node visited, a $DecisionWait$, is not added initially because of this (although its connection with node 1 is FC, a $DecisionWait$ is not one of the nodes without FC violations on any of its connections). Later on, when no more $Sequence$s, $Concur$s and $Synch$s can be added to the cluster, the $DecisionWait$ refereed before is again considered for inclusion (order 11) but then because of the previous inclusion of the $Concur$ visited with order 4, it can not be added. A different situation happens with the $DecisionWait$ on the top of the figure: it is the sixth node visited and is also ruled out initially, but at the end, when no more $Sequence$s, $Concur$s and $Synch$s can be added, it is inserted into the cluster because its connection with the already included nodes does not violate the FC conditions.

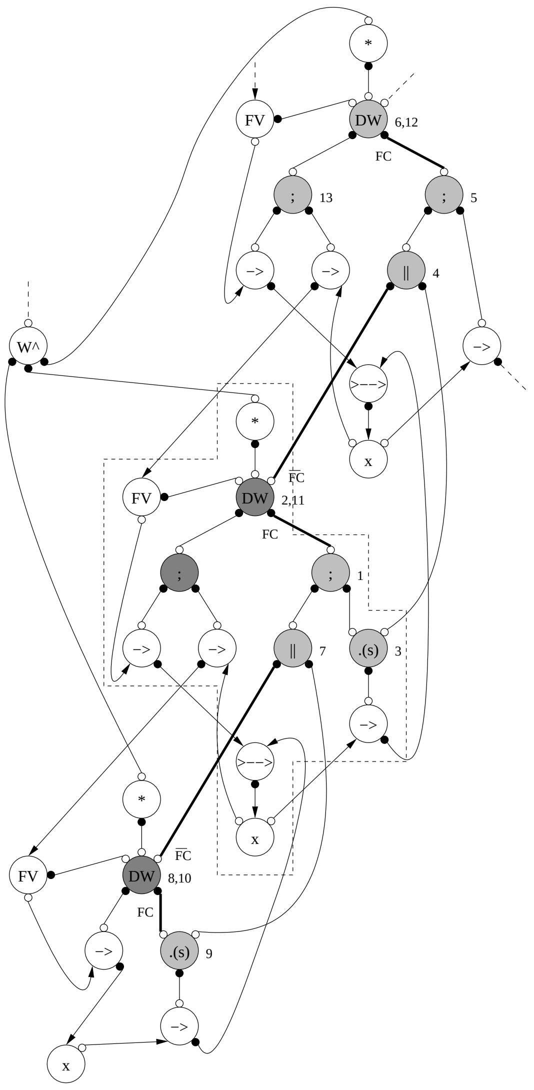


Figure 5. Stack Example

If the capacity of the $Stack$ grows, the HC set indicated with discontinuous lines in Figure 5 is repeated, due to its recursive specification. Then, in the FC mode, a main cluster (of $Sequence$s, $Concur$s and $Synch$s) and several satellite clusters (of one $Sequence$ and one $DecisionWait$) are obtained.

5.2. Example results

The $Stack$ example has been implemented and simulated for different capacities (4, 8, 12) and clustering modes (FC and AC). The area reduction and the performace improvement of the optimized clusters are presented in Figure 4. The Y-axis shows the percentage of reduction (im-

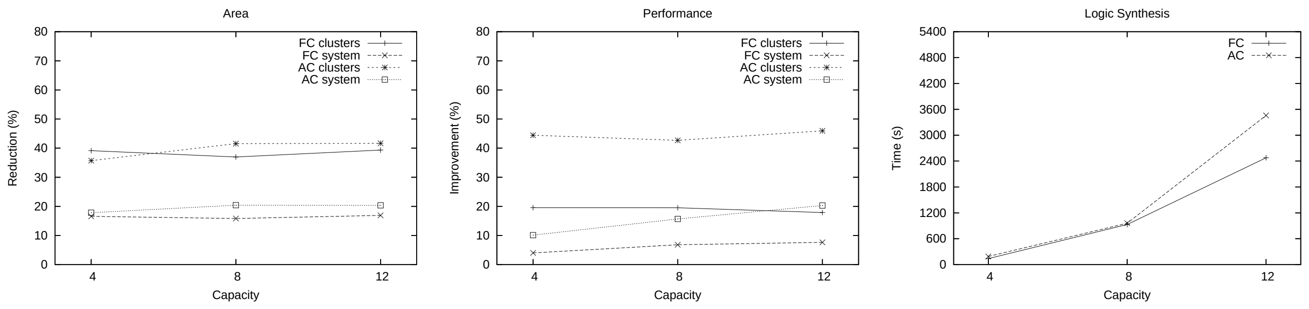


Figure 4. Stack Results

provement) and the X-axis the capacity of the $Stack$. The diverse line styles indicate the usage of the FC or AC mode and the results for the cluster or the system. The reduction in area is greater than 35% for the clusters and greater than 15% in the system. The performance improvement of the optimized clusters is nearly 20% for the FC mode and around 45% for the AC mode. This improvement for the system grows with the capacity, but in the AC mode this tendency is more significant than in FC mode. The time spent in logic synthesis grows with the capacity, but in the AC mode this increase is also more accelerated than in the FC mode.

6. Experimental Results

The theory described in this paper has been implemented into a back-end tool to support the logic synthesis of clustered handshake components. The tool can be used in different scenarios: a blind use would be to let the tool decide the clusters, by means of limiting the CPU time allowed for synthesis. For instance, the synthesis of FC nets leads to short CPU times in practice. The tool can also be forced to cluster a specific Petri net class, as shown in the algorithm in previous section. This later use is the one used in the experiments. Once a cluster is selected for optimization, the steps described in Section 3 are performed. We have used some of the Balsa examples provided with the tool: ArbTree, PopCount, Shifter and Stack.

6.1 Area results

For each example, we present two types of results. First, we show the area reduction in the clusters, and second we provide the impact of this improvement with respect to the overall system. For that purpose we estimate the area of a cluster by counting the area of its gates. The area of a gate g can be modeled with the following equation:

$$area(g) = \lambda_a(1 + log(fanin(g))$$

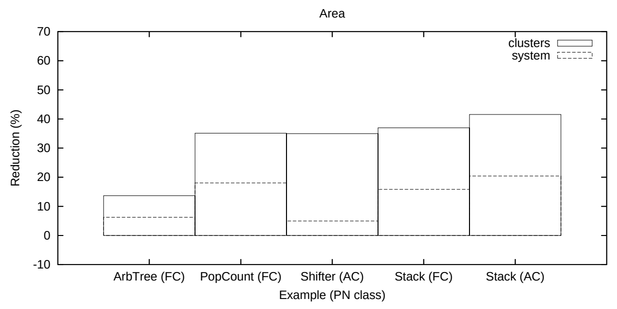
where λ_a represents the area of an inverter. This model gives an estimation of the complexity of an implementation depending on the number of gates it contains. The gates in this model are weighted by their fanin. Figure 6 shows the results in area. The Y-axis shows the percentage of reduction with respect to the clusters without optimization (continuous boxes) and the overall system[3] (discontinuous lines). X-axis shows the results for each benchmark used. Notice that for some examples, two results are presented, one for each Petri net class considered. In general, the significance in area reduction within the clusters (up to 40% in $Stack(AC)$) implies a significant area reduction within the system (up to 20% in $Stack(AC)$).

6.2 Performance results

Performance estimation has been done by adapting the Balsa simulator. As it was done for area results, we present the improvement in performance within the clusters and the influence of this improvement with respect to the overall system. The delay model used for a gate g is equal to the area model but with a different constant factor. With this delay model, three simulation times are found for each example: the elapsed time for the system (1) without optimization, (2) with optimized clusters and (3) with zero-delay clusters. The value $(1) - (3)$ represents the delay of the clusters without optimization, whereas $(2) - (3)$ is the delay of the optimized clusters. Figure 6 also shows the results in performance, where continuous boxes represent the performance improvement for the clusters and discontinuous lines represent the consequence of this improvement in the overall system.

The results on performance are not as uniform as in area. Excluding the first two examples (ArbTree and PopCount), the remaining examples can be optimized, but in general this improvement has a limited impact in the overall system, due to Amdahl's law [1]. For instance, for the Shifter example, the maximal improvement that could be achieved

[3]The area estimation provided by Balsa has been adapted to the model of this paper, in order to measure the area of non-clustered components.



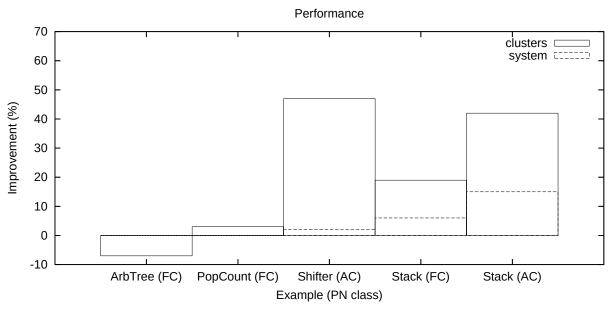


Figure 6. Balsa Examples Results

(the one obtained by using the zero-delay cluster) is less than 5%.

For the examples where the delay was degraded or maintained in the clusters (notice that also due to Amdahl's law this degradation was not transferred to the system), the signal insertion method could not find an appropriate way to insert state signals in a way that performance could be improved. See below for an explanation on this.

6.3 Discussion

The experimental results, specially in performance, could be improved by extending the theory presented in this paper into several dimensions: first, the set of HC considered for clustering (the ones in Table 2) can be extended to allow the clustering of more HCs. Second, complex HCs representing program modules has not been considered and its optimization could lead to significant improvements. This happens in the ArbTree example, where the potentially clusterizable HCs represent only a portion of the overall system. Third, the concurrent signal insertions to optimize performance in [7] may have an area penalty that induces significant delay overhead in some of the gates. This delays sometimes can not be compensated with the increase of the concurrency obtained.

7. Conclusions

A clustering technique to optimize the synthesis of HDL specifications has been presented in this paper. By using knowledge on the components of the HC network, the search can be guided to derive clusters for which the logic synthesis methods can be safely applied in practice. The underlying formalism used to represent a cluster is Petri nets, and the growing of a cluster can be controlled by using Petri net structural conditions. The approach has been implemented and integrated into the Balsa synthesis flow, and the preliminary experimental results obtained show significant improvements in area and, if the extensions suggested in this paper are incorporated, promising performance gains.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 79–81, 2000.

[2] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.

[3] C. H. K. v. Berkel, M. B. Josephs, and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, Feb. 1999.

[4] K. v. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[5] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, Apr. 2000.

[6] J. Carmona, J. M. Colom, J. Cortadella, and F. García-Vallés. Synthesis of asynchronous controllers using integer linear programming. *IEEE Transactions on Computer-Aided Design*, 25(9):1637–1651, Sept. 2006.

[7] J. Carmona and J. Cortadella. State encoding of large asynchronous controllers. In *Proc. ACM/IEEE Design Automation Conference*, pages 939–944, July 2006.

[8] T. Chelcea, A. Bardsley, D. Edwards, and S. M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 330–337, Mar. 2002.

[9] T. Chelcea and S. M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[10] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[11] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[12] F. Fernández. Logic synthesis of handshake components using clustering techniques. Master's thesis, Universitat Politècnica de Catalunya, June 2007.

[13] R. M. Fuhrer and S. M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Kluwer Academic Publishers, 2001.

[14] International technology roadmap for semiconductors: Design.
www.itrs.net/Links/2005ITRS/Design2005.pdf, 2005.

[15] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar. 1996.

[16] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[17] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[18] L. A. Plana, S. Taylor, and D. Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *ICCD*, pages 703–710. IEEE Computer Society, 2005.

[19] S. Taylor. *Data-Driven Handshake Circuit Synthesis*. PhD thesis, Dept. of Computer Science, University of Manchester, 2007.

[20] C. Ykman-Couvreur, B. Lin, and H. de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, Sept. 1994. User and Tutorial manual.

A. Enclosure Properties of Parallel Composition

In order to demonstrate the enclosure properties of the parallel composition, special parts of it will be used:

Definition 3 (Area Not Synchronized) *Given STGs G_1 and G_2, the area not synchonized of G_1 when it is composed whith G_2 is denoted by $NotSynch(G_1) = ((P_{NS1}, T_{NS1}, F_{NS1}, m_{0NS1}), \Sigma_{NS1}, \Lambda_{NS1})$, where:*

$$\begin{aligned}
&P_{NS1} = P_{NS1}^{pre} \cup P_{NS1}^{post}\\
&P_{NS1}^{pre} = \{(p_1,p_2) \in P_{||} \mid p_1 \in P_1, \forall (t_1,t_2) \in {}^\bullet(p_1,p_2),\\
&\qquad\qquad \Lambda_{||}((t_1,t_2)) \notin \Sigma_1 \cap \Sigma_2 \times \{+,-\}\}\\
&P_{NS1}^{post} = \{(p_1,p_2) \in P_{||} \mid p_1 \in P_1, \forall (t_1,t_2) \in (p_1,p_2)^\bullet,\\
&\qquad\qquad \Lambda_{||}((t_1,t_2)) \notin \Sigma_1 \cap \Sigma_2 \times \{+,-\}\}\\
&T_{NS1} = \{(t_1,t_2) \in T_{||} \mid \exists (p_1,p_2) \in {}^\bullet(t_1,t_2), (p_1,p_2) \in P_{NS1}^{post}\\
&\qquad\qquad \mathit{or}\ \exists (p_1,p_2) \in (t_1,t_2)^\bullet, (p_1,p_2) \in P_{NS1}^{pre}\}\\
&F_{NS1} = \{((p_1,p_2),(t_1,t_2)) \in F_{||} \mid (p_1,p_2) \in P_{NS1}^{post},\\
&\qquad\qquad (t_1,t_2) \in T_{NS1}\}\\
&\qquad \cup \{((t_1,t_2),(p_1,p_2)) \in F_{||} \mid (p_1,p_2) \in P_{NS1}^{pre},\\
&\qquad\qquad (t_1,t_2) \in T_{NS1}\}\\
&m_{0NS1}((p_1,p_2)) = m_{0||}((p_1,p_2))\\
&\Sigma_{NS1} = \{\sigma \in \Sigma_{||} \mid \exists (t_1,t_2) \in T_{NS1}, \Lambda_{||}((t_1,t_2)) \in \sigma \times \{+,-\}\}\\
&\Lambda_{NS1}((t_1,t_2)) = \Lambda_{||}((t_1,t_2))
\end{aligned}$$

Intuitively, $NotSynch(G_1)$ contains the flow relation between its places without shared transitions in their presets (postsets) and these presets (postsets).

Proposition 3 *Parallel composition areas:*

$$\begin{aligned}
G_1||G_2 \quad = \quad & NotSynch(G_1)\\
& \cup Synch(G_1, G_2)\\
& \cup NotSynch(G_2),
\end{aligned}$$

where the operator $\cup$ among STGs denotes the union of their sets.

Proof: It follows from the definitions of parallel composition (1), synchronization area (2), area not synchronized (3) and union of STGs. $\square$

Intuitively, $P_{||}$ is partitioned into P_{NS1}^{pre}, P_S^{pre} and P_{NS2}^{pre} (P_{NS1}^{post}, P_S^{post} and P_{NS2}^{post}) according to the existence of a shared transition in their presets (postsets). Given (p_1,p_2), if $(p_1,p_2) \in P_A^{pre}$ ($(p_1,p_2) \in P_A^{post}$), where $A \in \{NS1, S, NS2\}$, then ${}^\bullet(p_1,p_2) \subseteq T_A$ ($(p_1,p_2)^\bullet \subseteq T_A$) and ${}^\bullet(p_1,p_2) \times \{(p_1,p_2)\} \subseteq F_A$ ($\{(p_1,p_2)\} \times (p_1,p_2)^\bullet \subseteq F_A$). Therefore, $F_{||}$ is partitioned into F_{NS1}, F_S and F_{NS2}.

Proof of Proposition 1: Suppose G_1, G_2 and $Synch(G_1,G_2)$ are in C and $G_1||G_2$ is not in C. Using Proposition 3, $NotSynch(G_1) \cup Synch(G_1,G_2) \cup NotSynch(G_2)$ is not in C. Thus, there exists (p_1,p_2) and (p'_1,p'_2) in $P_{NS1}^{post} \cup P_S^{post} \cup P_{NS2}^{post}$ such that they violate C.

Given (p''_1,p''_2), if $(p''_1,p''_2) \in P_{NSi}^{post}$, where $1 \le i \le 2$, then for each $(t''_1,t''_2) \in T_{NSi} : ((p''_1,p''_2),(t''_1,t''_2)) \in F_{NSi}$ implies $(p''_i,t''_i) \in F_i$. If $(p''_1,p''_2) \in P_S^{post}$ and $p''_i \in P_i$ then for each $(t''_1,t''_2) \in T_S : ((p''_1,p''_2),(t''_1,t''_2)) \in F_S$ implies $(p''_i,t''_i) \in F_i$.

If $(p_1,p_2),(p'_1,p'_2) \in P_{NS1}^{post}$, since their flow relations are preserved, then there exists $p_1, p'_1 \in P_1$ such that they violate C, a contradiction. It is similar if $(p_1,p_2),(p'_1,p'_2) \in P_S^{post}$ or $(p_1,p_2),(p'_1,p'_2) \in P_{NS2}^{post}$.

If $(p_1,p_2) \in P_{NS1}^{post}$ and $(p'_1,p'_2) \in P_S^{post}$, as $(p_1,p_2)^\bullet \cap (p'_1,p'_2)^\bullet \neq \emptyset$, then $p'_1 \in P_1$. Since their flow relations are also preserved, there exists $p_1, p'_1 \in P_1$ such that they violate C, a contradiction. It is similar if $(p_1,p_2) \in P_{NS2}^{post}$ and $(p'_1,p'_2) \in P_S^{post}$.

If $(p_1,p_2) \in P_{NS1}^{post}$ and $(p'_1,p'_2) \in P_{NS2}^{post}$, as $(p_1,p_2)^\bullet$ and $(p'_1,p'_2)^\bullet$ do not contain any shared transition, then $(p_1,p_2)^\bullet \cap (p'_1,p'_2)^\bullet = \emptyset$, a contradiction.

Proof of Proposition 2: Suppose G_1, G_2 or $Synch(G_1,G_2)$ is not in C and $G_1||G_2$ is in C. Using proposition 3, $NotSynch(G_1) \cup Synch(G_1,G_2) \cup NotSynch(G_2)$ is in C. Thus, there is not (p_1,p_2) and (p'_1,p'_2) in $G_\cup$ such that (p_1,p_2) and (p'_1,p'_2) violate C.

The parallel composition only performs the Cartesian product of shared transitions. Therefore, if there is not (p_1,p_2) and (p'_1,p'_2) violating C in $NotSynch(G_1) \cup Synch(G_1,G_2)$ ($Synch(G_1,G_2) \cup NotSynch(G_2)$) then there is not (p_1,p_2) and (p'_1,p'_2) violating C in G_1 (G_2).