Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits *

Marco A. Peña and Jordi Cortadella
Department of Computer Architecture
Universitat Politècnica de Catalunya
08071-Barcelona, Spain
{marcoa, jordic}@ac.upc.es

Abstract

This paper presents a new methodology to automatically synthesize asynchronous circuits from descriptions based on process algebra. Traditionally, syntax-directed techniques have been used to generate a netlist of basic components previously implemented by skilled designers. However, the generality of the approach often involves the insertion of redundant functionality to the circuit.

We propose a new approach based on the composition of Petri nets and the automatic synthesis through Signal Transition Graphs that allows to take advantage of logic synthesis methods to optimize the circuit and make it portable for different delay models and technologies. Some preliminary experimental results have shown the effectiveness of the approach to improve the quality of the circuits.

1 Introduction

Process algebras have been successfully used for the specification and formal verification of digital asynchronous circuits. Several algebras based on the semantics of Hoare's CSP [9] and trace theory [22] have been proposed for different delay models [15, 2, 10, 7, 1].

In CSP-based algebras, the computation of a system is specified as a set of communicating processes that must be connected according to some discipline that guarantees a correct composition. Each language construct is hierarchically translated into a netlist of processes. The primitives of the language can describe parallel and sequential composition of processes, communication, synchronization and choice.

In models based on trace theory, the behavior of a circuit is specified as a set of traces of its environment alphabet. The model includes statements such as *concatenation* or *projection* to implicitly enumerate all possible strings of symbols generated by the functionality of the circuit.

Both models provide high-level abstractions of the behavior of a circuit. This is the main reason why synthesis is done through the so-called *syntax-directed translation* paradigm, consisting in hierarchically creating a netlist of atoms that implement the unrefinable primitives of the language. Handshake components such as "*parallelizer*" and "*mixer*" used for the translation of TANGRAM [20], modules like "*decision wait*", "*toggle*" and "*merge*" for delay-insensitive algebras [10], or the macromodules in [2] and [8] are some illustrative examples of basic primitives. Skilled designers must provide an efficient implementation for each module according to the semantics of the language.

Each basic module must be designed assuming that it must be able to interact with any correct environment. However, each particular composition of modules forces a different sequence of events of the environment of each of its components, thus often using a subset of their complete functionality. In some cases, composition patterns often used can be identified and substituted by cheaper implementations. This is the purpose of the peephole optimizations proposed in [20, 8]. Syntax-directed translation provides a completely automatic method for synthesis, although the logic obtained with such method may contain a significant degree of redundancy with regard to the required functionality.

Martin's technique based on the generation of production rules through handshake expansion [15] allows to synthesize a circuit at a finer granularity (gate or transistor level). However this method requires an expert designer's interaction for some decisions that have significant impact on the quality of the circuit (e.g. reshuffling and insertion of state variables for state disambiguation).

On another side, Petri net-based techniques for synthesis are totally automatic and have been proved to be efficient for moderate size descriptions [14, 13]. By using low-level synthesis tools, logic synthesis techniques to minimize combinational and sequential circuits as well as different delay models (e.g. bounded wire delays [13] or unbounded gate delays [12]) can be considered for the same input specification of a circuit. However, many designers agree in that describing the behavior of a circuit with a Petri net is an intricate task.

In this work we propose to combine both models for the synthesis of asynchronous circuits to benefit from the advantages of each method: process algebras provide a

*This work has been supported by the Ministry of Education of Spain (CYCIT TIC 95-0419) and the Departament d'Ensenyament de la Generalitat de Catalunya.

0-8186-7298-6/96 $5.00 © 1996 IEEE

neat formalism for describing behavior, whereas Petri nets play the rôle of the "assembly language" that describes the low-level operations to be executed.

An important part of this work will be devoted to propose the required link between process algebras and Petri nets. We will show how the abstraction provided by the basic modules is essential for the method. But rather than asking an expert designer to make an efficient circuit implementation of each module, we will ask the designer of the language to define the behavior of each module with a Petri net. Finally, given a netlist of basic modules and a library of Petri nets describing the basic modules, a circuit will be derived by the automatic composition of the Petri nets and the use of automatic logic synthesis tools.

The key advantages of this approach are the following:

- The designer of a language does not need to implement the basic modules of each construct. Only the behavioral description of such modules is required for synthesis.
- By using low-level synthesis techniques, the input description can be made portable for different delay models and technologies.
- The final circuit can be automatically optimized to strictly adjust its functionality to the behavior of its environment, thus eliminating the redundancy introduced by pure syntax-directed translation techniques.

2 Methodology

An overview of the methodology presented in this paper is depicted in Figure 1. From the designer's point of view, this framework is based on the well-known VLSI programming paradigm which looks at circuit design as a simple programming activity.

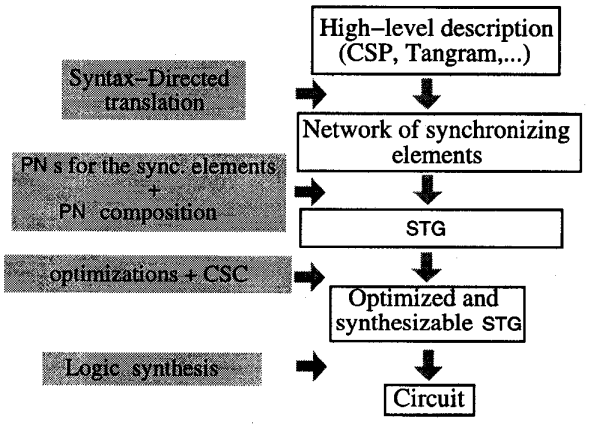
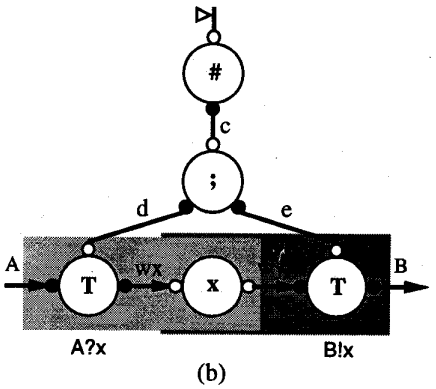


Figure 1: Overview of the synthesis methodology

In such approach the behavior of the system is described by a program in a high-level programming language, and the corresponding VLSI circuit is obtained automatically

```
( A:? bool & B:! bool ).
|[
    x: var bool
|
    #[ A?x ; B!x ]
]|
```

(a)



(b)

Figure 2: One-place buffer: (a) TANGRAM program, (b) network of handshake components

by a compiler. Thus, the efforts of the designer will be focused on the algorithmic and architectural aspects of the system, while the low-level and physical aspects will be addressed by the compiler.

A program consists of a set of concurrent processes which cooperate by the synchronization and exchange of messages along channels. Its translation into a circuit is done by *syntax-directed* translation (SDT), where the compiler generates a network of simple asynchronous synchronizing elements that implement the primitives of the language. Such elements are already implemented in a library.

A programming language very similar to TANGRAM is used in our framework. The basic components implemented for each language construct are called *handshake components* [20].

The TANGRAM program in Figure 2 (a) describes a one-place buffer [20]. Its translation into *handshake components* is depicted in Figure 2 (b). It consists of 5 handshake components, 5 channels (c, d, e, wx and wy) and 3 ports ($\triangleright$ or activation port, **A** and **B**). A given channel connects one passive port (white circle) to an active one (black circle). The communication along a channel is by means of a simple four-phase[1] handshake protocol in which the active port starts the synchronization.

2.1 Petri nets and their composition

When two synchronizing elements are connected, their original behaviors are mutually restricted to satisfy their

[1] Two-phase protocols can also be supported

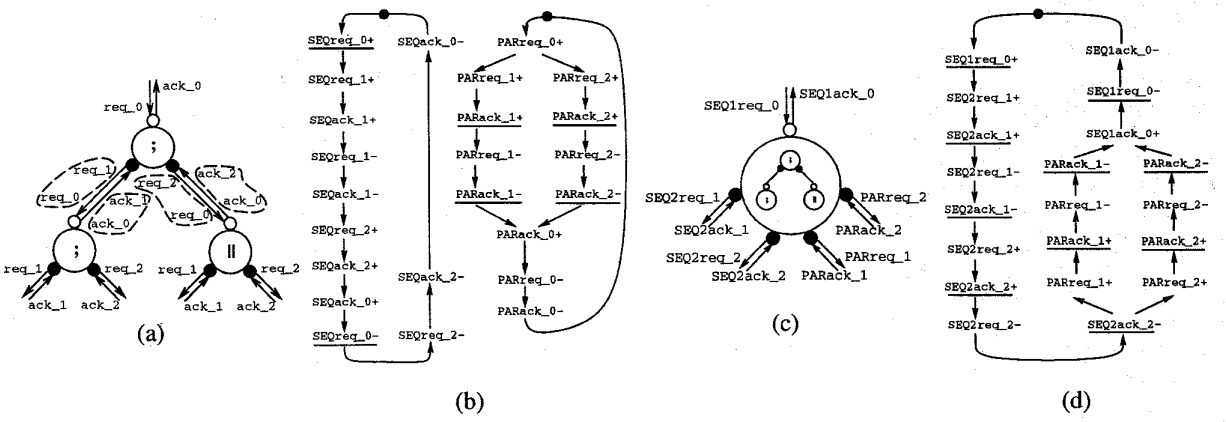


Figure 3: Connection of handshake components: (a) network of components, (b) STGs for a *sequencer* and a *parallelizer*, (c) new macro-component and (d) equivalent STG for the entire circuit.

synchronizations. Therefore, we may expect that some simplifications can be done in the corresponding sub-circuits when they interact with others.

In our methodology we propose to *automatically* create new *macro-handshake* components by synthesizing a Signal Transition Graph (STG) [18, 3] obtained from the composition of the STGs of the basic components.

Figure 3 illustrates how the composition of STGs is performed according to the connectivity of the handshake components of a circuit. Figure 3 (b) describes the STGs of a *sequencer* and a *parallelizer* respectively. After composing the two *sequencers* and the *parallelizer* as shown in Figure 3 (a), and after removing the internal events of the new macro-handshake component (Figure 3 (c)), the STG of Figure 3 (d) is obtained.

It is important to emphasize that the composition of STGs and the re-synthesis of the new STG is done completely automatically. The re-synthesis of STGs is performed by `petrify` [5], a tool for Petri net synthesis. The method for Petri net composition is presented in Section 4.

2.2 Petri net re-synthesis

`Petrify` [5] is a novel tool that allows to synthesize a Petri net from either another Petri net or a state graph. This tool has played a key rôle in the optimization of the STGs obtained by composition.

Within our synthesis framework, and after constructing a Petri net by composition, `petrify` performs the following tasks:

1. builds a transition system by performing a token flow analysis of the Petri net. Symbolic techniques (BDD-based) are used to handle Petri nets with a vast space of markings.

2. eliminates those dummy and internal events that have appeared as the result of the composition (projects the transition system onto the external events).

3. re-synthesizes a new simplified Petri net from the transition system obtained after the projection.

Moreover, a new algorithm for state encoding has been integrated in `petrify` [4] that allows to directly obtain an implementable STG. We refer the reader to [5] for further details on the synthesis of Petri nets from transition systems.

3 Modeling processes with Petri nets

In this section we provide some terminology and definitions related to formalisms we will use later. This includes specification and composition of processes, Petri nets and how they model a process.

3.1 Alphabets, traces and processes

Definition 3.1 (symbol, alphabet, trace)
An **alphabet** *is a finite set of* **symbols**. *Symbols are denoted by identifiers. Finite sequences of symbols are called* **traces**. Σ^* *denotes the set of all traces over alphabet* Σ, *including the empty trace* λ.

Definition 3.2 (concatenation, projection, ϵ)
Concatenation *is denoted by the '$\cdot$' operator, where the empty trace* λ *is the neutral element.*
The **projection** *of a trace* s *on an alphabet* Σ, *denoted by* $s\lceil\Sigma$ *is defined as follows:*

$$\lambda\lceil\Sigma = \lambda$$

$$(s_o \cdot s)\lceil\Sigma = \begin{cases} s\lceil\Sigma & \text{if } s_o \notin \Sigma \\ s_o \cdot (s\lceil\Sigma) & \text{otherwise} \end{cases}$$

*A new special "*silent*" symbol* ϵ *is defined, for which* $s = \epsilon \cdot s = s \cdot \epsilon$.

Definition 3.3 (process) *[17]*
A **process** P *is the pair* $P = (\alpha P, \beta P)$ *where* αP *stands for an alphabet of* **events**, *and* $\beta P \subseteq \alpha P^*$ *is the set of traces modeling the behavior of* P.

Definition 3.4 (parallel composition of processes)
Given two processes $P = (\alpha P, \beta P)$ *and* $Q = (\alpha Q, \beta Q)$ *their* **parallel composition** $P \parallel Q$ *is defined by:*

$$\begin{aligned} \alpha(P \parallel Q) &= \alpha P \cup \alpha Q \\ \beta(P \parallel Q) &= \{s \in \alpha(P \parallel Q)^* \mid (s\lceil \alpha P) \in \beta P \wedge (s\lceil \alpha Q) \in \beta Q\} \end{aligned}$$

The new set of traces reflects the interaction between processes P *and* Q*. The traces of both processes are "*synchronized*" on the common events, if they exist, i.e.* $\alpha P \cap \alpha Q \neq \emptyset$.

We are mainly interested in modeling such composable processes with labeled Petri Nets, and in defining new parallel composition operators in this domain.

3.2 Petri nets

Definition 3.5 (labeled Petri net)
A **labeled Petri net** *is a 6-tuple* $N = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$*, 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 a finite set of* **arcs** *representing the* **flow relation**, $M_o : P \rightarrow I\!N$ *is the initial* **marking** *(state) of the Petri net,* Σ *is an* **alphabet**, *and* $\Lambda : T \rightarrow \Sigma \cup \{\epsilon\}$ *is a* **labeling function**.

Definition 3.6 (pre-set, post-set, transition firing)
Given a labeled Petri net $N = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$*:*

(a) The **pre-set** *and* **post-set** *of a node* $x \in P \cup T$ *are denoted by* ${}^\bullet x = \{y \mid (y, x) \in F\}$ *and* $x^\bullet = \{y \mid (x, y) \in F\}$*, respectively.*

(b) A transition $t \in T$ *is* **enabled** *in a marking* M*, denoted by* $M[t\rangle$*, when all places in* ${}^\bullet t$ *are marked. This is* $\forall p \in {}^\bullet t\ ,\ M(p) \geq 1$.

(c) An enabled transition t *in marking* M*,* **fires** *removing a token from places in* ${}^\bullet t$ *and adding a token to places in* $t^\bullet$*, reaching a new marking* M' *(*$M[t\rangle M'$*), i.e.:*

$$\forall p \in P\ ,\ M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) & \text{otherwise} \end{cases}$$

(d) A marking M *is* **reachable** *from* M_o *if there is a* **firing sequence** *of transitions* $\vec{t} = t_1 t_2 \ldots \in T^*$ *that transforms* M_o *into* M*, i.e.* $M_o[\vec{t}\rangle M$*. The set of all reachable markings from* M_o *is denoted by* $[M_o\rangle$.

Definition 3.7 (language of a Petri net)
Given a labeled Petri net $N = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$*, we denote by* $\Pi(N)$ *the set of all possible firing sequences over* T*:*

$$\Pi(N) = \bigcup_{M \in [M_o\rangle} \{\vec{t} = t_1 t_2 \ldots \in T^* \mid M_o[\vec{t}\rangle M\}$$

The **language** $L(N)$ *over the alphabet* Σ *is given by:*

$$L(N) = \{s = s_1 s_2 \ldots \in \Sigma^* \mid \exists \vec{t} = t_1 t_2 \ldots \in \Pi(N) \wedge \forall i, s_i = \Lambda(t_i)\}$$

Definition 3.8 (modeling of processes)
A labeled Petri net $N = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$ **models** *the process* $P = (\alpha P, \beta P)$ *if* $\Sigma = \alpha P$ *and* $L(N) = \beta P$.

4 Parallel composition of Petri nets

This section presents a new definition for the parallel composition of Petri nets. This definition keeps the equivalence with respect to the behavior expressed by the parallel composition of processes.

The definition given here is intended to be used within the framework described in Section 2. We consider Petri nets modeling the behavior of synchronizing elements (i.e. handshake components in the case of TANGRAM). The circuit generated by the compiler is correct by construction, and such elements follow an established synchronization protocol which guarantees the correctness of their composition.

The method we present has linear cost on the number of transitions, places and arcs of the original Petri nets. Such linearity is achieved because of the restrictions on the STGs we are composing, and the use of dummy (silent) transitions in the construction of the synchronization areas. Moreover, in most practical cases many of the places and silent events added by the algorithm can be removed by local transformations during the composition process itself. In contrast, other previous approaches ([17, 6]) require quadratic transition splitting and a subsequent merging of the pairs of transitions generated for every common symbol.

Another approach for Petri net composition (*product* of two Petri nets) is presented in [23]. This is a very liberal form of parallel composition of Petri nets in which arbitrary synchronizations are allowed, i.e. synchronizations between common events may or may not occur nondeterministically. Moreover, the model assumes that each transition represents a different event. The previous assumptions make this composition approach unappropriate for our framework.

4.1 Composable Petri nets

When defining the synchronizing elements that will be connected later to build a circuit, some restrictions about their behavior must be assumed.

For sake of simplicity, in the sequel we will refer to definitions and properties given for the handshake circuits [20]. However, they also apply for any asynchronous architecture under the VLSI programming paradigm, like those in [15, 7].

Firstly we must define the correctness of the individual behaviors of the handshake components (i.e. *handshake processes*). Therefore, we must charecterize the *port structure* of these elements and the *traces* of communication events they can produce through their ports. A handshake through a port basically consists of two events: a request followed by an acknowledgement. In this sense, we consider traces in which the occurrence of requests and acknowledgements of each port strictly alternate and in which the first event of each port is a request (see Definition 2.8 in [20]). In terms of Petri nets these consideration can be interpreted like the switchover and non-autoconcurrency conditions required for the synthesis of speed-independent circuits ([11]).

On the other hand we must talk about what conditions make the handshake circuits to be suitable for connection

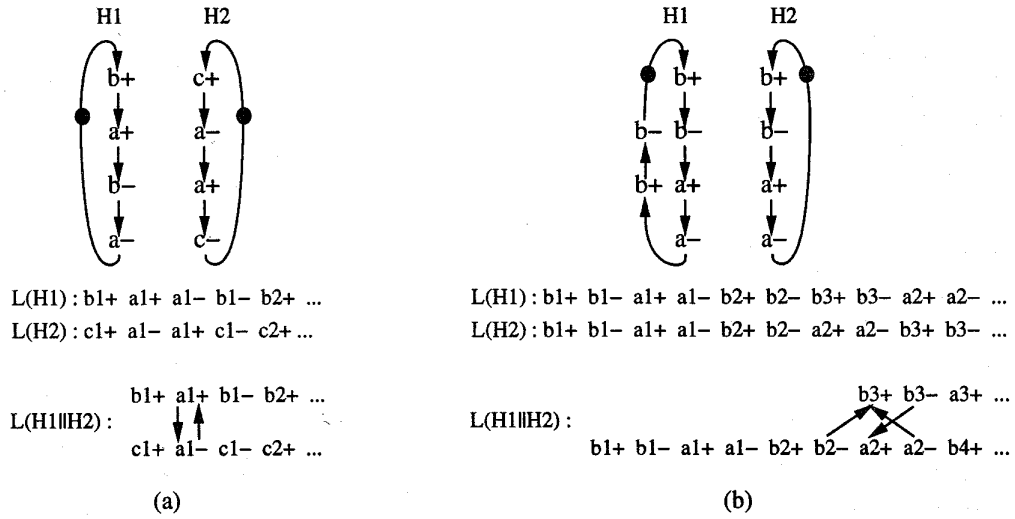


Figure 4: Two simple examples of incorrect composition: due to the incompatibility between the ordering of common events (a), and due to the mismatch of the multiplicity of the common events (b)

with others, in such a way that the result will be a new handshake circuit with equivalent behavior. In this sense an important fact is that the parallel composition is only defined for *connectable* handshake processes. Connectablility of handshake circuits establishes that a pasive port can only be connected to a single active port and vice versa (see Definition 3.0 in [20]). Another key fact is the required absence of *interference*. Interference with respect to symbols occurs when one process sends a symbol and the other process is not ready to receive it. The *receptiveness* of handshake processes and the imposed handshake protocol exclude the possibility of interference (see Section 3.0 in [20]). This means that no deadlock can be produced in the synchronization between handshake components because of their parallel composition. And this can be directly translated in terms of the Petri nets which will model their behaviors.

All these conditions make the circuit to be correct by construction as stated in [20]. In this way, the behavior of the handshake components can be described using Petri nets in a simple way, and to derive correct algorithms to perform the parallel composition of the hanshake components in terms of the Petri nets which model their behaviors.

Figure 4 shows two simple examples of incorrect composition. In the first case, an incompatible ordering of the common events $a+$ and $a-$ leads to a deadlock. In the second case, when $H1$ has completed a full cycle and is in its initial state waiting to fire $b+$, the token in $H2$ is in the arc '$b- \rightarrow a+$', and a deadlock is produced. This kind of situations are not allowed by the conditions stated above.

In the sequel, only composable Petri nets joining the above requirements will be considered.

4.2 Parallel composition of Petri nets

Definition 4.1 (T^σ)
Given a labeled Petri net $N = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$, *and a symbol* $\sigma \in \Sigma$ *the set*

$$T^\sigma = \{t \in T \mid \Lambda(t) = \sigma\}$$

contains all the transitions labeled with the same event name σ.

Let $N_1 = \langle P_1, T_1, F_1, M_{o1}, \Sigma_1, \Lambda_1 \rangle$ and $N_2 = \langle P_2, T_2, F_2, M_{o2}, \Sigma_2, \Lambda_2 \rangle$ be two labeled Petri nets such that $(P_1 \cup T_1) \cap (P_2 \cup T_2) = \emptyset$, i.e. they are node-disjoint. Their parallel composition is denoted by $N = N_1 \parallel N_2$.

The composition is defined by the construction of a so-called *synchronization area* for every common event $\sigma \in \Sigma_1 \cap \Sigma_2$. Hence we define the way the synchronization area is created for each of such events. We will consider the most general case in which $\mid T_1^\sigma \mid = m \geq 1$ and $\mid T_2^\sigma \mid = n \geq 1$. That is, there is an arbitrary number of transitions labeled with σ in both Petri nets.

Construction of the synchronization area

In the process of building a synchronization area for a given $\sigma \in \Sigma_1 \cap \Sigma_2$, some new places and transitions are created, while others are removed. The same occurs with the flow relation. The following sets are defined:

- $P_\sigma = \{p_u\ ,\ p_d\} \cup \mathcal{E}P_\sigma$, where:
  - p_u and p_d are two new places added to perform the synchronization between both Petri nets.
  - $\mathcal{E}P_\sigma = \bigcup_{t \in T_1^\sigma} \{p_t\}$ is the set of new places created to keep the relation among predecessors and successors of each transition of T_1^σ.

- $\mathcal{E}T_\sigma = \mathcal{E}U_\sigma \cup \mathcal{E}D_\sigma$ is the set of new silent events created to hide the transitions of T_1^σ, where

$$\mathcal{E}U_\sigma = \bigcup_{t \in T_1^\sigma} \{\varepsilon_{t,u}\} \quad \text{and} \quad \mathcal{E}D_\sigma = \bigcup_{t \in T_1^\sigma} \{\varepsilon_{t,d}\}$$

- $RF_\sigma = \{(p,t) \in F_1 \mid t \in T_1^\sigma\} \cup \{(t,p) \in F_1 \mid t \in T_1^\sigma\}$ is the set of flow relations removed from F because of the removal of transitions in T_1^σ.

- $$AF_\sigma = \bigcup_{t \in T_1^\sigma} \{(p, \varepsilon_{t,u}) \mid \varepsilon_{t,u} \in \mathcal{E}U_\sigma \wedge p \in {}^\bullet t\} \cup \bigcup_{t \in T_1^\sigma} \{(\varepsilon_{t,d}, p) \mid \varepsilon_{t,d} \in \mathcal{E}D_\sigma \wedge p \in t^\bullet\} \cup \bigcup_{t \in T_1^\sigma} \{(\varepsilon_{t,u}, p_t) \mid \varepsilon_{t,u} \in \mathcal{E}U_\sigma \wedge p_t \in \mathcal{E}P_\sigma\} \cup \bigcup_{t \in T_1^\sigma} \{(p_t, \varepsilon_{t,d}) \mid \varepsilon_{t,d} \in \mathcal{E}D_\sigma \wedge p_t \in \mathcal{E}P_\sigma\}$$

  is the set of new flow relations added to F instead of those in RF_σ.

- $SF_\sigma = (\mathcal{E}U_\sigma \times \{p_u\}) \cup (\{p_u\} \times T_2^\sigma) \cup (T_2^\sigma \times \{p_d\}) \cup (\{p_d\} \times \mathcal{E}D_\sigma)$ is the set of new flow relations created to join both Petri nets by the synchronization places.

The idea behind the synchronization area is that the transitions of both Petri nets will fire concurrently until the Petri nets reach a marking "*willing to enter*" this area. At this point the synchronization will take place. Once it has occurred both Petri nets will continue their concurrent evolution until another synchronization area is reached. Moreover, the traces of events of the new Petri net will only reflect the firing of the t'_j transition (labeled with σ), because the rest of the transitions in the synchronization area are silent events (i.e. they produce the empty trace). An example of how such synchronization area for a given $\sigma \in \Sigma_1 \cap \Sigma_2$ is built, may be seen in Figure 5.

Definition 4.2 (parallel composition of Petri nets)
The new Petri net $N = N_1 \parallel N_2 = \langle P, T, F, M_o, \Sigma, \Lambda \rangle$ *after the creation of the synchronization areas for every* $\sigma \in \Sigma_1 \cap \Sigma_2$ *is defined as follows:*

- $P = P_1 \cup P_2 \cup \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} P_\sigma$

- $T = (T_1 \setminus \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} T_1^\sigma) \cup T_2 \cup \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} \mathcal{E}T_\sigma$

- $F = (F_1 \setminus \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} RF_\sigma) \cup F_2 \cup \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} (AF_\sigma \cup SF_\sigma)$

- $M_o : P \to \mathbb{N}$, *is defined as:*

$$\forall p \in P, M_o(p) = \begin{cases} M_{o1}(p) & \text{if } p \in P_1 \\ M_{o2}(p) & \text{if } p \in P_2 \\ 0 & \text{otherwise} \end{cases}$$

- $\Sigma = \Sigma_1 \cup \Sigma_2$

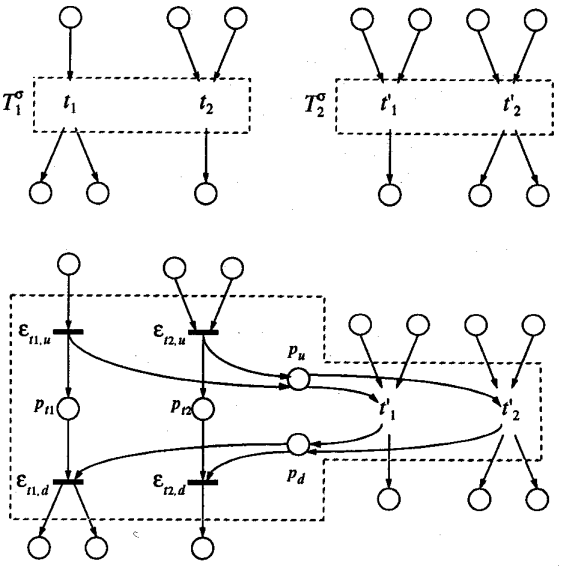


Figure 5: Construction of the synchronization area for a common symbol $\sigma \in \Sigma_1 \cap \Sigma_2$

- $\Lambda : T \to \Sigma \cup \{\epsilon\}$ *is defined as:*

$$\forall t \in T, \Lambda(t) = \begin{cases} \Lambda_1(t) & \text{if } t \in T_1 \setminus \bigcup_{\sigma \in \Sigma_1 \cap \Sigma_2} T_1^\sigma \\ \Lambda_2(t) & \text{if } t \in T_2 \\ \epsilon & \text{otherwise} \end{cases}$$

Note that all the new silent events created for a given $\sigma \in \Sigma_1 \cap \Sigma_2$, $\mathcal{E}T_\sigma$ *set, are labeled with* ϵ.

Even this composition method requires the insertion of silent events, for most practical cases we have encountered in our experiments, we have seen that the silent events can be easily removed by local transformations of the Petri net. As a last resort, `petrify` removes all those silent events that could not be eliminated after the composition.

Figure 6 shows an example of composition of two simple Petri nets. We may see the Petri net once the synchronization area for event a is built (Figure 6 (b)). After some local transformations to remove the silent events and places introduced by the composition algorithm, a simpler Petri net is obtained (Figure 6 (c)).

Theorem 4.3 (Composition theorem)
Let N_1, N_2 *be two labeled Petri nets. Then:*

$P(N_1 \parallel N_2) = P(N_1) \parallel P(N_2)$, *i.e.:*

(a) $\alpha P(N_1 \parallel N_2) = \alpha(P(N_1) \parallel P(N_2))$

(b) $\beta P(N_1 \parallel N_2) = \beta(P(N_1) \parallel P(N_2))$

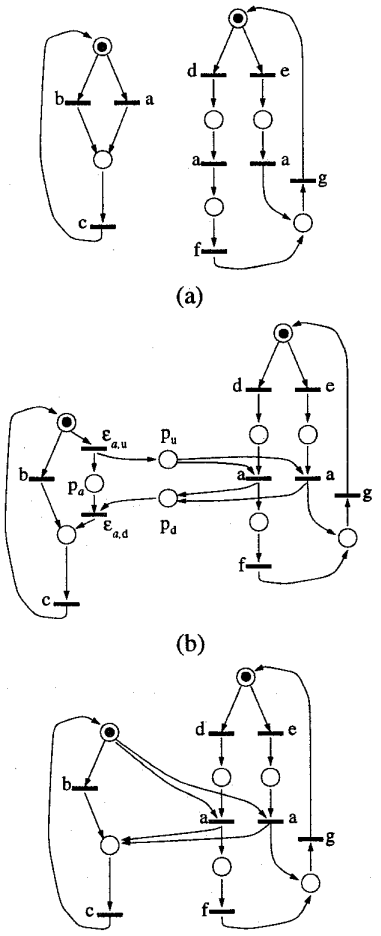


Figure 6: Given two Petri nets (a): parallel composition before (b) and after (c) the elimination of silent events

Proof:

The alphabet equivalence (a) is an obvious consequence of Definitions 4.2 and 3.4.

We now present an intuitive proof for equivalence (b). Rather than proving trace equivalence for the composition of Petri nets, we merely prove that "traversing" the synchronization area only generates the observable symbol σ. Assuming this fact, a complete proof by induction on the length of the traces can be easily derived (see [16]), since the behavior of the Petri net outside the synchronization area is identical to the parallel behavior of the original Petri nets.

Assume we have a marking M in which some transition of the synchronization area is enabled (say $\varepsilon_{t_1,u}$ in Figure 5). By the construction of the synchronization area we have that

$$\exists t \in T_1^\sigma : \; M[\varepsilon_{t,u}\rangle \wedge \varepsilon_{t,u} \in \mathcal{E}U_\sigma$$

The firing sequence $M[\varepsilon_{t,u} \cdot t' \cdot \varepsilon_{t,d}\rangle M'$ can take place, where M' is the marking obtained after traversing the synchronization area. Moreover, the marking M' is identical to the composition of the markings of the original Petri nets (N_1 and N_2) after having fired a transition from T_1^σ and T_2^σ respectively.

The transition sequence $s = \varepsilon_{t,u} \cdot t' \cdot \varepsilon_{t,d}$ has only the symbol σ as observable trace, i.e. $\Lambda(s) = \Lambda(\varepsilon_{t,u}) \cdot \Lambda(t') \cdot \Lambda(\varepsilon_{t,d}) = \epsilon \cdot \sigma \cdot \epsilon = \sigma$. That is, the only observable event inside a synchronization area corresponds to the one that will occur in the synchronization of the related processes. □

Property 4.4 (Linear cost)
Let $N_1 = \langle P_1, T_1, F_1, M_{o1}, \Sigma_1, \Lambda_1\rangle$ and $N_2 = \langle P_2, T_2, F_2, M_{o2}, \Sigma_2, \Lambda_2\rangle$ be two labeled Petri nets. Their parallel composition $N = N_1 \parallel N_2$ has linear cost on the number of transitions, places and arcs of N_1 and N_2.

Proof:

The proof focuses on how the construction of the synchronization areas for the common symbols affects the cardinality of the sets of places, transitions and arcs.

Let us consider the most general (worst) case of composition for a given $\sigma \in \Sigma_1 \cap \Sigma_2$, where $\mid T_1^\sigma \mid = m_\sigma \geq 1$ and $\mid T_2^\sigma \mid = n_\sigma \geq 1$, i.e. the Petri nets can contain more than one transition labeled with the same symbol. Note that $\sum_{\sigma \in \Sigma_1 \cap \Sigma_2} m_\sigma \leq \mid T_1 \mid$ and $\sum_{\sigma \in \Sigma_1 \cap \Sigma_2} n_\sigma \leq \mid T_2 \mid$.

The following expressions show the cardinality of the sets P, T and F in the new Petri net $N = N_1 \parallel N_2$:

$$\mid P \mid = \mid P_1 \mid + \mid P_2 \mid + \sum_{\sigma \in \Sigma_1 \cap \Sigma_2} (2 + m_\sigma)$$

$$\mid T \mid = \mid T_1 \mid + \mid T_2 \mid + \sum_{\sigma \in \Sigma_1 \cap \Sigma_2} m_\sigma$$

$$\mid F \mid = \mid F_1 \mid + \mid F_2 \mid + \sum_{\sigma \in \Sigma_1 \cap \Sigma_2} (4m_\sigma + 2n_\sigma)$$

The terms in the expressions above, represent the amount of information required to perform the composition between two Petri nets. The cost of the composition is linear on the number of transitions, places and arcs of the original Petri nets.

Note that the number of implementable transitions (i.e. silent events) has been reduced from $\mid T_1 \mid + \mid T_2 \mid$ to $\mid T_1 \mid - \sum_{\sigma \in \Sigma_1 \cap \Sigma_2} \mid T_1^\sigma \mid + \mid T_2 \mid$.

A full proof of this property and a complete derivation of the expressions given above can be found in [16]. □

5 Experimental results

This section presents the experimental results obtained by the application of our methodology to some TANGRAM examples.

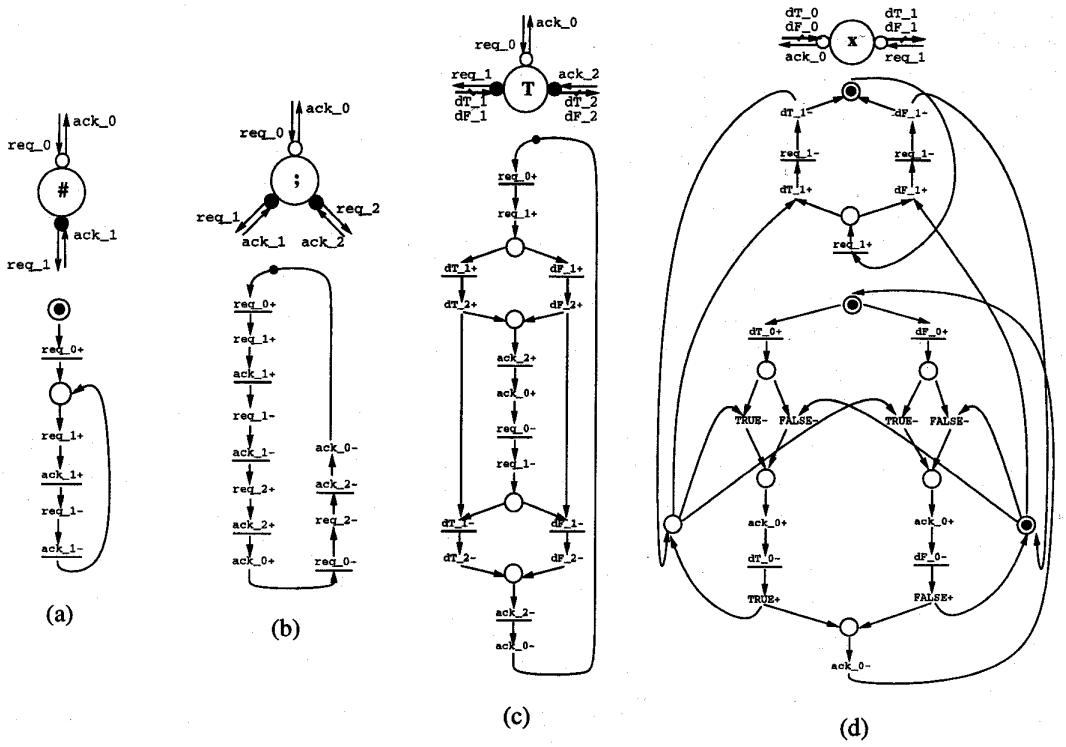


Figure 7: STGs for the different handshake components composing the One-place buffer

5.1 The synthesis system

The methodology presented in this paper is implemented by a set of automatic tools (see Figure 1). Our first tool is a syntax-directed translator from a TANGRAM-like language into a network of handshake components. The behavior of each handshake component has been described by means of an STG.

The central tool transforms a network of synchronizing elements into a single STG with equivalent behavior. It uses the library of STGs and connects the several descriptions which forms the circuit, using the Petri net composition algorithm described in Section 4.2.

Then `petrify` is used to remove the internal events produced in the composition process, and to perform some optimizations on the new global STG. It is also used to make the new STG satisfy the CSC condition. Finally, a speed-independent circuit is synthesized using SIS [19].

5.2 Circuit partitioning

The complexity manageable by current synthesis tools is directly constrained by the size of the state graph of the circuit. For this reason it is not always possible to handle the STG we obtain for the complete input description. Therefore, partitioning techniques have to be devised to fully automate the synthesis.

Our synthesis tools cannot partition the circuit automatically yet. This is an important part of our future work to complete the framework. In some of the presented examples, partitioning has been required to obtain a final implementation. The number of handshake components that can be currently considered for a feasible synthesis ranges between 5 and 20, depending on the behavior and connectivity of the network.

Two basic criteria have been used to partition the network of handshake components: minimize the interconnection between partitions and minimize the degree of parallelism inside each partition. Thus we try to derive the largest partitions whose corresponding STGs can be handled by the subsequent state-based tools, i.e. `petrify` and the logic synthesis tools.

We expect the size of the STGs handled by future synthesis tools will increase by using new symbolic or structural syntesis techniques. If so, larger partitions or even the full STG generated for the complete input description could be handled, and thus better area results could be achieved.

5.3 A complete example

To illustrate our methodology we have chosen the well-known One-place buffer [20]. A TANGRAM program describing its behavior is shown in Figure 2 (a). The program is translated into a network of handshake components by means of syntax-directed translation (Figure 2 (b)). The STGs describing the behavior of each handshake compo-

nent are shown in Figure 7. After Petri net composition an STG for the whole circuit is obtained (see Figure 8 (a), in this case all silent events have been automatically removed by local transformations on the STG). `Petrify` has been used to remove the internal events, re-synthesize the STG (see Figure 8 (b)) and solve state encoding by inserting state signals (Figure 8 (c)). The synthesized circuit is depicted in Figure 8 (d).

5.4 Results

Most of the benchmarks are classical in TANGRAM literature. Here we report the results obtained for some buffers (BUF1, RIP_BUF, WAG_BUF) and shift register elements (SRC, SRD) [20], a 3-token FIFO, a DME arbiter cell and the XYZ benchmark (modeled in TANGRAM from its original STG description). The BUF1 example is the one depicted in Figure 2.

Table 1 summarizes the results. The SDT columns reports the number of handshake components and the area obtained by the pure syntax-directed approach. The next column reports the number of states of the STG for the whole circuit. When partitioning has been required, the states of the STG for each partition are shown. The area results obtained using our methodology are listed in the next column (the `asynch` library of SIS has been used). The last column presents the area reduction obtained with our method. The area reported for the pure syntax-directed approach has been calculated by implementing the handshake components according to the circuits described in [20] and using the same gate library.

It can be observed that, even the circuits obtained with our method have been derived automatically, significant area reductions can still be achieved. We expect to improve these results in the future by

- improving the quality of the synthesis tools for STGs and
- increasing the size of the circuits manageable by the synthesis tools by using structural and/or symbolic methods to represent the space of states of the STGs.

The XYZ example is one of the classical benchmarks used in the literature on STGs. Our TANGRAM-like programming language has been modified to allow the description of interfaces with individual signals also, rather than channels that implicitly require two handshake signals. New non-handshake components to link the external signals with the internal handshake components have been derived (similar components have been presented in [21]). The complexity added by the new components and the unsuitability of TANGRAM to describe timing-diagram-like behaviors is the main reason of the drastic difference of both implementations. This example corroborates the effectiveness of the Petri net composition and re-synthesis method: from a network of 12 handshake components, an STG with 3 signals, 6 transitions and 8 states was derived.

	SDT		final STG		Red.
Example	HSKs	Area	States	Area	(%)
BUF1	5	402	18	192	52
RIP_BUF	12	1076	9+247	786	27
WAG_BUF	16	1992	16+564+112	1906	4
SRC	8	914	43+19080	768	16
SRD	11	1414	95+4460+612	1328	6
3-FIFO	14	1112	53	870	22
DME	13	550	52+392+332	500	10
XYZ	12	1686	8	72	95
TOTAL		9146		6422	30

Table 1: Experimental results

6 Conclusions

We have presented a new methodology for the synthesis of VLSI asynchronous circuits from high-level specifications. Behaviors described with process algebras are translated into a network of synchronizing elements by means of syntax-directed translation. We have used the Petri net formalism to build a library of behavioral descriptions for such elements.

A new Petri net composition method with linear cost with respect to the number of transitions, places and arcs of the original Petri nets has been proposed. The composition method is complemented with an optimization process which removes the new internal and silent events originated on the composition. As a result, a single STG with equivalent external behavior is derived for the whole network of synchronizing elements.

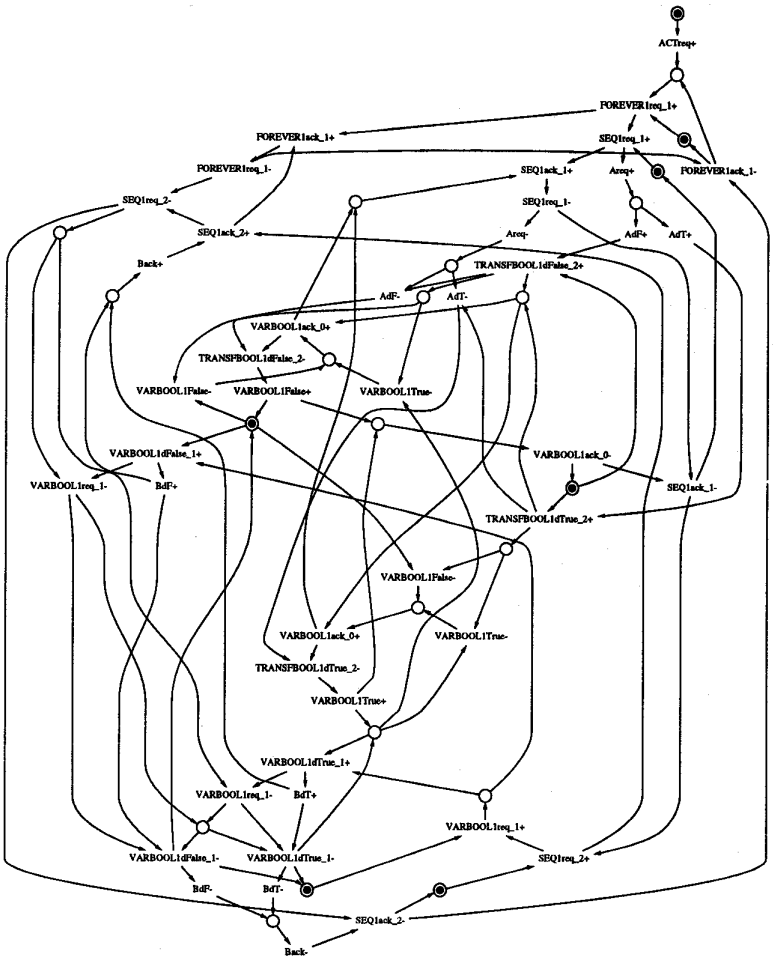
The circuit is then synthesized by CAD tools. Significant improvements with regard to methods using pure syntax-directed translation can be obtained.

Classical CAD tools are state-based tools. The exponential cost of building the state graph restricts the size of the synthesizable circuits with these tools. Therefore, in some of the benchmarks we must partition the network of handshake components and apply the methodology to each partition separately. We expect that the size of the STGs handled by future synthesis tools will drastically increase by using synthesis techniques at structural level or by using symbolic methods. If this happens, the quality of the circuits will still be further improved.

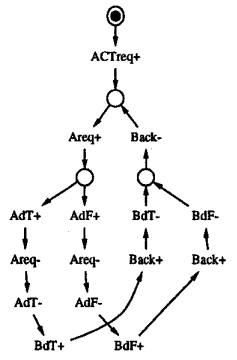
We must remark that all the tasks involved in this methodology are fully automated by several tools[2]. The methodology presented in this paper can be used for the synthesis of asynchronous circuits described with any language based on communicating processes, such as CSP, TANGRAM, OCCAM, etc. Currently, our framework is able to synthesize any netlist of handshake components as far as a behavioral description (STG) for each of them is provided.

Finally, the combination of process algebras and Petri nets for modeling asynchronous circuits opens a new route for verification. We do not discard the possibility of using unfoldings or symbolic traversal of Petri nets for the formal verification of circuits whose environment can be specified with process algebras.

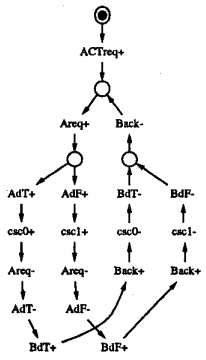
[2]Only automatic partitioning is under development



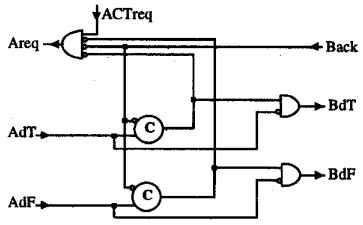
(a)



(b)



(c)



(d)

Figure 8: One-place buffer: (a) STG derived automatically by Petri net composition and re-synthesis (the picture is the graphical output automatically generated by `petrify`). (b) STG obtained after removing internal events. (c) STG after solving CSC. (d) Synthesized circuit.

References

[1] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 587–591. IEEE Computer Society Press, November 1992.

[2] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 262–265. IEEE Computer Society Press, November 1989.

[3] T.-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. of the IEEE International Conference on Computer Design*, pages 220–223. IEEE Computer Society Press, October 1987.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on theory of regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Aizu, Japan, March 1996.

[5] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri Nets from State-Based Models. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 164–171. IEEE Computer Society Press, November 1995.

[6] G. de Jong and B. Lin. A Communicating Petri Net Model for the Design of Concurrent Asynchronous Modules. In *Proc. ACM/IEEE Design Automation Conference*, pages 49–55, June 1994.

[7] J. C. Ebergen. *Translating programs into delay-insensitive circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[8] G. Gopalakrishnan, P. Kudva, E. Brunvand, and V. Akella. Peephole optimization of asynchronous macromodule networks. In *Proc. of the IEEE International Conference on Computer Design*, pages 442–446. IEEE Computer Society Press, October 1994.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1989.

[10] M. B. Josephs and J. T. Udding. An algebra for delay-insensitive circuits. In Robert P. Kurshan and Edmund M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.

[11] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[12] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 56–62, June 1994.

[13] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[14] B. Lin, C. Ykman-Couvreur, and P. Vanbekbergen. A general state graph transformation framework for asynchronous synthesis. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 448–453. IEEE Computer Society Press, September 1994.

[15] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[16] M. A. Peña and J. Cortadella. Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits. Technical Report RR-95/48, UPC/DAC, 1995.

[17] I. Reicher and M. Yoeli. Net-based modeling of communicating parallel processes with applications to VLSI design. Technical Report 532, Technion-Israel Institute of Technology, December 1988.

[18] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, July 1985.

[19] 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 circuits synthesis. Technical Report M92/41, UCB/ERL, May 1992.

[20] K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[21] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[22] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[23] G. Winskel. Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation*, 7:197–238, 1987.