Journal of Circuits, Systems and Computers, Vol. 8, No.1 (1998) 67-118

The Use of Petri Nets for the Design and Verification of Asynchronous Circuits and Systems

Alex Kondratyev
The University of Aizu
Aizu-Wakamatsu, 965-80 Japan

Jordi Cortadella
Univ. Politecnica de Catalunya
Barcelona, Spain

Michael Kishinevsky
The University of Aizu
Aizu-Wakamatsu, 965-80 Japan

Luciano Lavagno
Politecnico di Torino
10129 Torino, Italy

Alexander Taubin
The University of Aizu
Aizu-Wakamatsu, 965-80 Japan

Abstract

Petri nets [46, 37, 45, 48] are a powerful formalism for modeling concurrent systems. They are capable of implicitly describing a vast state space by a succinct representation which gracefully captures the notions of causality, concurrency and conflict between events. Petri nets have also been chosen by many authors as a formalism to describe the behavior of asynchronous circuits by interpreting the events as signal transitions, thus coining the term *Signal Transition Graph* (STG) [50, 4].

A design framework for asynchronous systems involves three main aspects: formal specification, verification and synthesis. In this paper we review the main techniques we have used to cover these aspects in recent years, with a special focus on asynchronous circuits.

1 Introduction

The synchronous hypothesis, stating that computation and communication take zero time and that events can only happen at some discrete points in time, is quite rigid and does not match well physical reality. However, it has enabled digital circuit design to make an incredible progress in recent years. The asynchronous hypothesis, stating that you cannot make any precise assumption on how much time computation and/or communication take, is (apparently) much more flexible and closer to reality. Yet, its role in digital circuit design is still marginal. We believe that the main reason for this is the lack of a widely accepted and well-established design flow for asynchronous circuits based on a sound mathematical model exactly like Finite State Machines and Boolean Algebra are the foundations that have made CAD-supported synchronous digital design possible.

The purpose of this paper is to review recent results in asynchronous circuit analysis and synthesis based on a sound mathematical model that yield efficient automated tools. Even though it is not, by itself, sufficient to bring asynchronous circuits (again) into mainstream logic design, since this would require a deep change in the way the design is carried out, we hope that it will help to promote the idea that asynchronicity is not synonymous of "black magic" any longer.

The mathematical model that we use is an *interpretation* of the widely known Petri Net (PN) model, in which every PN transition models a circuit signal change. We will show, by means of examples, how real (and fictitious, but well-known) problems can be specified in this form, and how they can be solved, both in terms of analyzing the specification properties, and in terms of generating a logic circuit implementation.

The class of asynchronous circuits that we consider in this paper is called Speed-independent (from the pioneering work of Muller [36, 34]). In this class, computation takes an a priori unbounded amount of time, while communication is assumed to take a small (more precisely, smaller than any computation) amount of time. This may appear unrealistic for modern VLSI technology, where communication across a large chip can actually take much longer than any gate switching. However, we consider it useful because:

- any point-to-point (single receiver) communication can be modeled as a computation delay without loss of generality,
- it is possible to enforce communication protocols between subcircuits that ensure no dependency on communication delays (such protocols are generically termed delay-insensitive [53, 35]),
- it enables the use of (restricted, as it will be described in Section 8) Boolean optimization to efficiently implement the circuit.

Without this hypothesis, optimization capabilities (and hence the applicability of asynchronous logic altogether) for delay-insensitive circuits are almost non-existent, and synthesis techniques amount to little more than a syntax-driven translation from a specification language and peephole optimizations.

Let us consider the use of STGs in the design flow for asynchronous circuits (see Figure 1).

Formal specification of asynchronous circuits STGs are used as a behavior specification of a system in a *top-down* design flow (block S1 in Figure 1). The STG model inherits the asynchronous semantics of PNs and can be naturally used for specifying behavior of the target asynchronous implementation. This approach is illustrated in the paper by considering several design examples.

PNs can be also used in a *bottom-up* design flow (block S2). For this, each circuit module (netlist of gates) can be syntactically translated into a PN, called a *circuit* PN[58]. This PN can be used for further verification and for circuit optimization via resynthesis.

Verification of asynchronous circuits Verification is used at different stages of design (the corresponding blocks in Figure 1 are marked by $V1$, $V2$ and $V3$). After a system is specified one should answer the following question: "Can it be implemented with an asynchronous circuit?" This requires verification of implementability properties (block $V1$ in Figure 1).

There is an iterative loop of changing the specification until it becomes implementable. While changing the specification the designer can use the information about violation of implementability properties that is provided by the verification process.

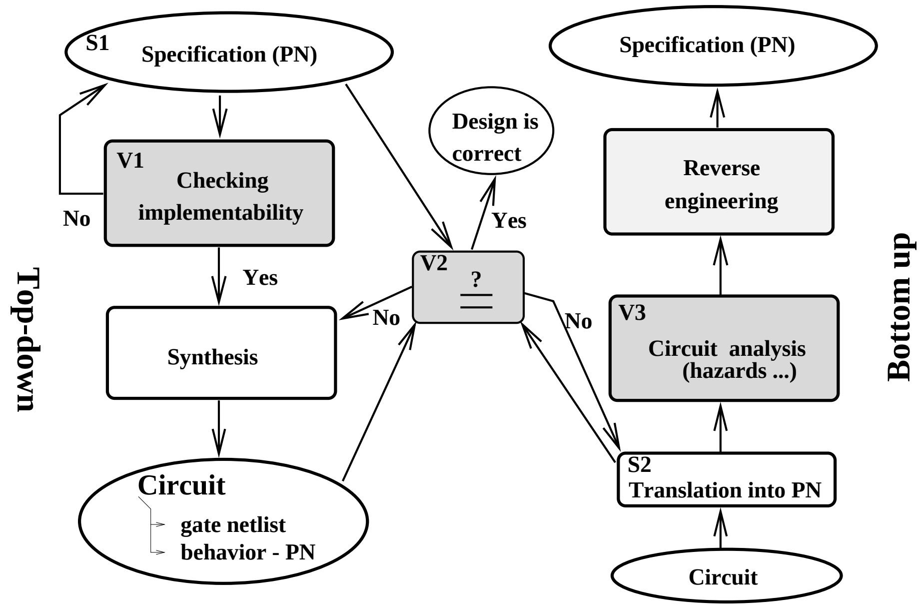


Figure 1: PN usage in design flows

The implementability conditions for STGs are known from literature [4, 28, 55]), where they have been formulated in terms of PN markings. Such an approach requires reachability analysis of a PN or an STG, using explicit generation of a reachability graph (or its binary encoded version - a State Graph), which can be very complex for a highly concurrent behavior.

There are several well-known techniques to fight with the "state explosion problem" in behavioral analysis of Petri Nets. Symbolic Binary Decision Diagram-based (BDD) traversal of a reachability graph allows its implicit representation which is generally more compact than an explicit enumeration of states [42]. Other methods avoid the generation of the corresponding reachability graph [32, 13, 22, 26] by considering a finite prefix of the equivalent occurrence net. This prefix, called an unfolding, is an acyclic net where all places have no more than one input transition.

In the paper, we will mainly concentrate on these two approaches for property verification using PNs although for verification of certain properties they can be complemented by the stubborn sets [54] and partial order techniques [16]. Other approaches for asynchronous circuits verification using PNs or similar formalisms can be found in [11, 1, 49, 61].

After the circuit is synthesized, the implementation verification is required for checking if the implementation (the circuit) conforms to the specification (block $V2$ in Figure 1) [11, 49]. In a PN-framework, this problem can be viewed as a comparison of two PNs: a PN describing the specification and a circuit PN corresponding to the implementation.

The implementation verification is used for hand-crafted designs, for automatically synthesized circuits if the synthesis method is not correct-by-construction, for debugging synthesis tehniques and for checking that the physical implementation actually corresponds to the design assumptions that are used in correct-by-construction synthesis methods. In the following, this task will be considered as a comparison of two PNs: one representing the specification and one derived from the circuit. For a correct design, the circuit PN must conform to the specification.

The circuit implementation may also require to verify some properties. For example, any asynchronous circuit should be free of hazards, that are short spurious pulses at the outputs of gates (block $V3$ in Figure 1). This can be done in the same way as in *top-down* design (module

$V1$) by translating the circuit specification into a corresponding PN [1].

Synthesis and optimization of asynchronous circuits Given a Signal Transition Graph, one can produce a net-list of an asynchronous controller in the target gate library while preserving the specified input-output behavior.

There are two different routes for synthesis. The first approach is a syntax-directed translation of a PN (verified for certain implementability properties beforehand) into a circuit. The direct translation approach is easy to implement but, in most practical cases, it would hardly yield an elegant and compact solution. The paper shortly discusses the direct translation methods before going into a more challenging synthesis approach (for more details on structural methods we refer to the survey [60]).

An alternative synthesis procedure is, in a way, similar to classical logic design of finite state machines. It can be seen as a set of equivalent transformations of the original specification (e.g., by means of event insertion) to satisfy the implementability properties. It includes state assignment by solving the Complete State Coding problem [4, 5] coupled with logic minimization and speed-independent technology mapping to a target library [51, 3, 25]. Related approaches can be also found in [56, 1, 30]. Methods based on the theory of state regions [12, 40, 6] provide an efficient framework for the solution of the above tasks, as they allow us to treat uniformly large sets of markings. State regions are also well suited for symbolic manipulation. These methods are discussed in Section 8.

2 Basic Notions

Let $N = \langle P, T, F, m_0 \rangle$ be a Petri net (PN) [37], where P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and m_0 is the initial marking.

A transition $t \in T$ is enabled at marking m_1 if all its input places are marked. An enabled transition t may fire producing a new marking m_2 with one less token in each input place and one more token in each output place (this is denoted by $m_1 \xrightarrow{t} m_2$ or $m_1 \rightarrow m_2$). The new marking m_2 can again make some transitions enabled. We can therefore talk about sequences of transitions that fire under the markings reachable from the initial marking m_0. Such sequences of transitions will be called *feasible traces* or simply *traces*. The set of input places of transition t is denoted by $\bullet t$ and the set of output places by $t\bullet$. Similarly, $\bullet p$ and $p\bullet$ stand for the sets of input and output transitions of place p. A place p is called a *choice place* if it has more than one output transition. A choice place is called *free-choice* if any of its output transitions has only one input place. A PN is *free-choice* if all its choice places are free-choice.

The set of all markings reachable in N from the initial marking m_0 is called the Reachability Set of N. Its graphical representation is called the Reachability Graph (RG).

An example of PN and of its reachability graph is shown in Figure 2,a,b. There are two choice places in the PN, $p1$ and $p5$, both of them are free-choice places and consequently the PN is free-choice. Since in every reachable marking of this PN there is only one token, each marking of its reachability graph contains only one place. This PN has no concurrency.

A PN is called:

- *k-bounded*, if for every reachable marking the number of tokens in any place is not greater than k (the place is called k-bounded if for every reachable marking the number of tokens

[1] Although verification by analyzing with partial order techniques the State Graph derived from a circuit can be more efficient in practice.

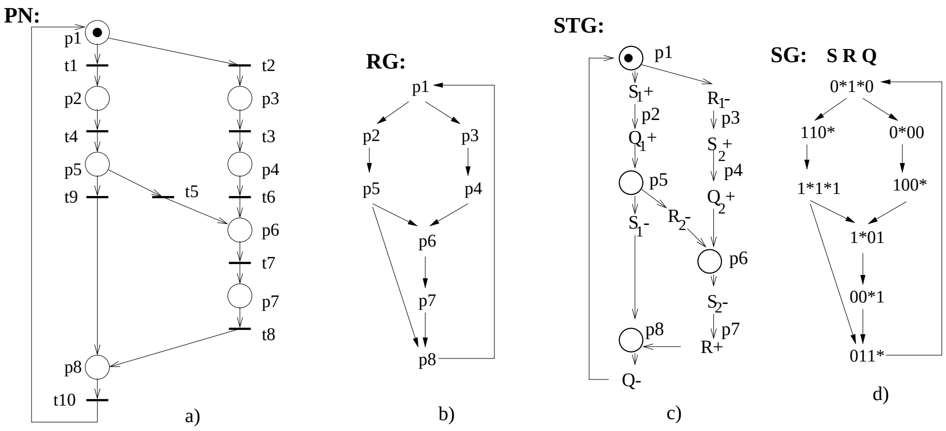


Figure 2: Example of PN and STG

in it is not greater than k),

- *bounded*, if there is a finite k for which it is k-bounded,
- *safe*, if it is 1-bounded (a 1-bounded place is called a safe place)

A transition t_i is called *non-persistent* if t_i is enabled in a reachable marking m together with another transition t_j, and t_i becomes disabled after firing t_j. Non-persistency of t_i with respect to t_j is also called *a direct conflict* between t_i and t_j. A PN is *persistent* if it does not contain any non-persistent transition.

In the PN in Figure 2,a transitions $t1$ and $t2$ are both enabled in the initial marking. Firing each of them disables the other, hence this PN is non-persistent.

To show the application of PNs to digital circuit design, we will consider the signal interpretation of PNs called Signal Transition Graphs. This model is widely used in asynchronous design for specifying control circuits [4, 28, 33].

Signal Transition Graphs (STGs) are PNs whose transitions are interpreted as changes of circuit signal levels. A signal transition can be represented by a_j+ for the j-th transition of signal a from 0 to 1 or a_j- for its j-th transition from 1 to 0, while a_j* is a generic name for either a rising or falling transition of a. S_A denotes the set of all signals of an STG that can be divided into input, output and internal signals. [2]

Definition 2.1 *An STG D is a triple $\langle N, S_A, \lambda\rangle$, where N is a PN, S_A is the set of signals and $\lambda : T \to S_A \times \{1, 2, \ldots\} \times \{+, -\}$ is the* partially *defined labeling function. A transition t with an undefined label, $\lambda(t)$, is called a* dummy *transition.*

This definition of an STG differs from the definition [4] in two ways. It does not limit the class of PNs to free-choice PNs and allows the use of dummy transitions.

An example of an STG is shown in Figure 2,c. This STG is obtained by labeling transitions of the PN from Figure 2,a with signal transitions. STGs are often drawn in their shorthand form, where transitions are denoted by their labels (instead of bars) and places with only one input and output transition are omitted.

[2] One can follow the origin of STG model from change maps [34] and signal graphs [21, 50]. It is worthwhile to mention also the related models like change diagrams [22] and labeled marked graphs [61].

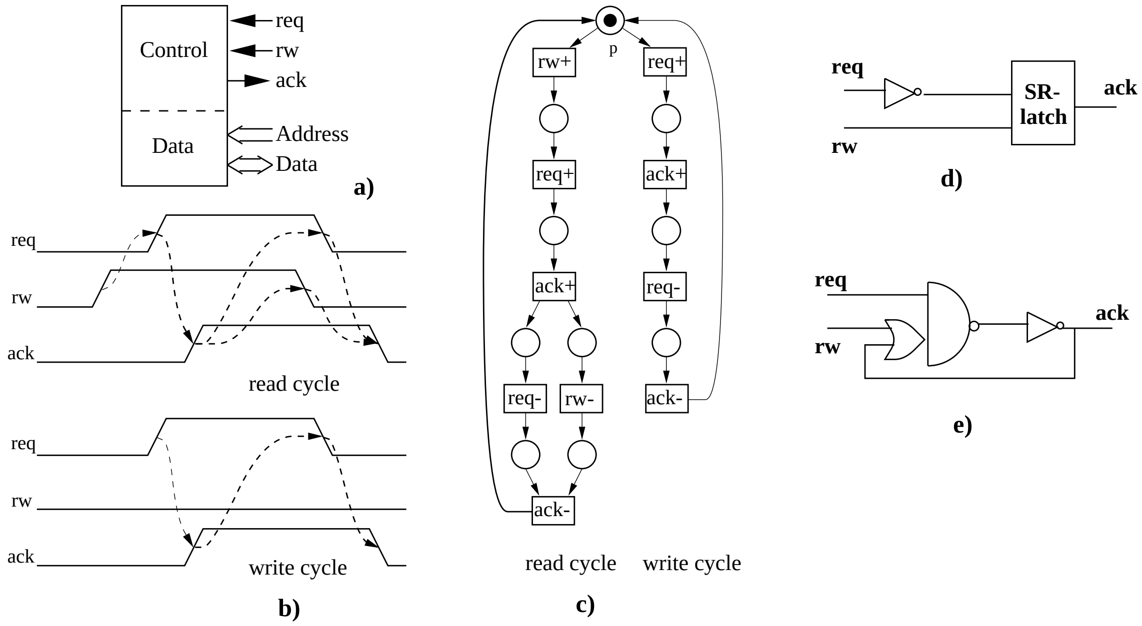


Figure 3: Specification of an asynchronous controler using Petri nets

The interpretation of PN transitions by the changes of binary signals allows us to introduce a binary encoded Reachability Graph for STGs. This graph is called a *State Graph (SG).* Historically, SGs appeared independently (and actually earlier) than PNs. They were used as a formal language in the pioneering work of D. Muller [36] to specify the behavior of the so-called speed-independent circuits. However, defining an SG as a supplementary model for an STG (as its encoded Reachability Graphs) gives nice opportunities to use both event-based (STG) and state-based models (SG) in a design flow. The advantages of this approach will become clear in the next sections.

An SG is a directed graph whose vertices correspond to the markings of the Reachability Graph. An SG vertex is labeled with a Boolean vector $s = \langle s^1, \ldots, s^n \rangle$ representing the value of the STG signals, where n is the number of signals in the STG. This vector is called a *state.* Two states s_1 and s_2 corresponding to markings m_1 and m_2 are connected with an edge in the SG if m_2 is reachable from m_1 by the firing of some event a_i* of the STG ($s_1 \overset{a_i*}{\to} s_2$). This transition a_i* is called *enabled* in state s_1. Signal a is called enabled in state s if some transition a_i* is enabled in s, otherwise a is called *stable* or *disabled.*

An example of SG for the STG in Figure 2,c is shown in Figure 2,d. The * marks the enabled signals in each SG state.

3 Overview of design methodology

This Section presents a simple example illustrating the design of asynchronous circuit by using interpreted Petri Nets. It is an informal introduction to the methodology that is discussed in more detail in further sections.

Figure 3 shows a behavioral specification of an asynchronous bus controller. The timing diagrams for the read and write cycles describe the temporal behavior in terms of signal transitions (events).

The diagrams show different types of relations between the events, namely

choice between the read and write cycle when the controller is in its initial state,

causality between pairs of ordered events such as $req+ \rightarrow ack+$ and

concurrency between pairs of events, such as $req-$ and $rw-$ in the read cycle, that can occur in any order.

Petri nets are appropriate to capture the previous relations in the same behavioral model. An interpreted Petri net (Signal Transition Graph) for the controller specification is shown in Figure 3,c.

In the implementation of controller we will follow the conventional scheme, separating the design of the datapath from the design of the control part. The datapath is a regular part, which is generally built by composing reusable units such as registers, ALUs, multipliers, etc. These units, and their asynchronous composition, require different design methods that we will not consider in this paper (see, e.g., [52]). On the other hand, the controller is generally much more irregular and difficult to reuse or assemble efficiently from basic components larger than standard logic gates. Hence the logic design of control parts is the main target of the methodology presented in this review.

Let us illustrate the design steps by means of the bus controller of Figure 3.

- **Specification** (see Section 4).

  At this stage we must formalize our notion of what we are going to implement. The output of the specification step is an STG describing the control part, like that in Figure 3,c.

- **Verification** (see Sections 6 and 7).

  Verification generally means to gain confidence that a formal specification does not contain any error. While absolute confidence can never be obtained, due to the impossibility to model all errors, our confidence in their absence is increased by checking simple properties, such as: no new request can be produced until the acknowledgement is done, the set phase (rising of request and acknowledgement) is followed by the reset phase (falling of request and acknowledgenment), etc.

  These properties can be analysed via the construction of a binary encoded rechability graph (called State Graph, see Section 2). It is easy to check that the State Graph of the controller in Figure 3 confirms the correctness of original specification.

- **Synthesis** (see Section 8)

  A State Graph that satisfies the implementability conditions, as checked during the verification step, defines the truth table for every signal of the STG. Hence we could use conventional logic minimization techniques to derive logic equations for all non-input signals. However these logic equations correspond to arbitrary boolean functions that may not be available in a given gate library. Therefore the functions should be decomposed on simpler components which can be directly matched with library gates. The latter step is called *technology mapping* and is non-trivial in asynchronous design, because decomposition may introduce hazards in the circuit implementation.

  The only non-input signal of the controller in Figure 3 is signal ack, which has a logic equation $ack = req + ack{*}rw$. Two different implementations of ack obtained by speed-independent technology mapping are shown in Figure 3,d,e.

4 Specification of circuits

First, we will show the use of PNs in a top-down design methodology when the design flow starts from a behavior specification and ends up with a circuit that implements it. This approach will be illustrated with several examples that will be used throughout the paper.

After this we will discuss how to use PNs in a bottom-up design flow. In this flow the starting point is a set of library blocks, and extracting from them a behavior specification can be useful for verification, optimization and resynthesis.

Set-dominant latch. A set-dominant latch is a basic block of logic design. Similar to a conventional SR-latch it has two inputs: S (set) and R (reset) and one output Q. The output Q goes high whenever the set input S is asserted ($S = 1$) and goes low when the reset input is asserted while the set input is low ($R = 1, S = 0$) (see Figure 4). Note that due to asymmetry in the latch behavior any combination of input values is feasible in a latch operation and this latch is more robust than the conventional SR-latch for which the combination $S = R = 1$ is forbidden. It is easy to see that the behavior specified by the STG in Figure 2,c can be implemented using a set-dominant latch. From this STG the logic equation for the latch output Q can be derived.

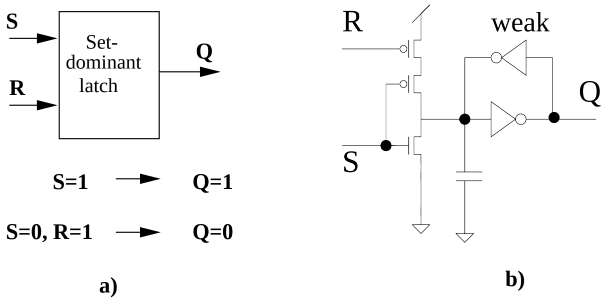


Figure 4: Set-dominant latch a) and its implementation b)

Consumer-producer problem. Let us consider the implementation of a control for a consumer-producer system with multiple consumers and a single producer. Consumers are synchronized with a producer by a buffer of restricted capacity. In the simplest case when the capacity of a buffer equals to 1, each consuming process requires one piece of production in a buffer and the producer does not start working again until the piece is consumed (the buffer is empty). The latter means that such system is operating only when there are requests on consuming, and is idle if no request is present.

Input-output behavior of a control unit for the consumer-producer system is specified by signals: $reqi$ – input requests for consuming from i-th consumer and con, $prod$ – output grants for consuming and producing respectively. We assume that only one consumer can produce a request $reqi+$ at any given time, and that the choice between requesting consumers is non-deterministic. This means that we *abstract* away from the specification any reason that the environment may have to decide between consumers. Abstraction and non-determinism are powerful mechanisms to simplify analysis and synthesis tasks of complex concurrent systems.

The control unit communicates with the environment in a handshake fashion. After a consumer has produced a request $reqi+$, it is granted by $con+$. Then the granted consumer removes the request ($reqi-$), and the control unit waits the synchronization with the producer ($prod+$) to report about the finishing of consuming ($con-$) and to restart the process from the initial state.

The process for a two-consumer system can be specified by the STG shown in Figure 5,a. Note that $req1+$ and $req2+$ are rising transitions of two distinct request signals, while con_1+ and con_2+ are transitions of the same signal. This means that several distinct PN transitions can have the same physical interpretation.

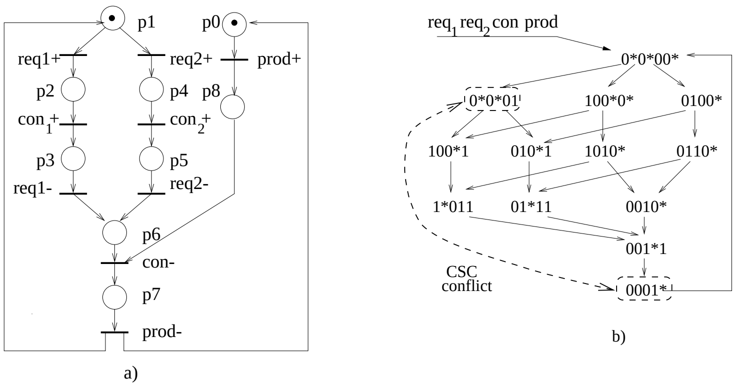


Figure 5: STG for a consumer-producer system a) and its SG b)

For the purpose of optimization, we would like to share the logic controlling different consumers as much as possible. Hence the last part of handshake communication (event $con-$) is made common for all processes. This approach is clearly different from a distributed control implementation where this event would be duplicated in each consumer process.

The implementation of the control unit for a consumer-producer system will be discussed in later sections.

Railway control. Consider an imaginary ring railway with several sections and two trains on them (see Figure 6). The trains move in only one direction and transport the goods. Since their speed and time for loading/unloading are not constrained in any way, it is necessary that appropriate control should guarantee safety of the railway. For that, let us assume that while the trains are in motion there must always be at least one free section between them. Since the delay of the processes are arbitrary (according to the problem semantics), the control circuit must be asynchronous and we will design it using STGs.

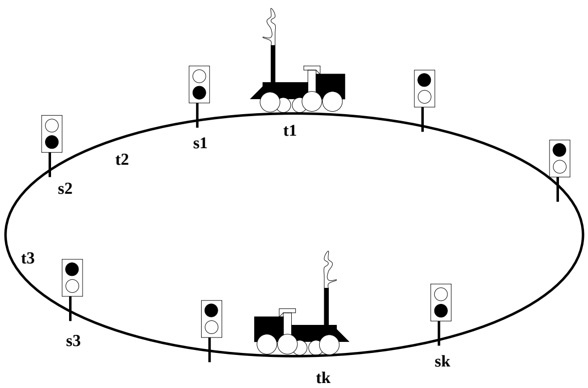


Figure 6: A ring railway with two trains

Let signals $T = \{t1, t2, \ldots\}$ denote the location of the trains on the sections 1, 2, ... ($ti = 1$ if there is a train on the i-th section and $ti = 0$ if no train is there) and signals $S = \{s1, s2, \ldots\}$

denote the states of the semaphore on the i-th section ($si = 1$ if the semaphore is open and $si = 0$ if it is closed). The task of the control circuit is to produce the output signals $s1, s2, \ldots$ controlling the semaphores, based on the values of detectors $t1, t2, \ldots$ and on the current state of the semaphores.

Let us derive the causal relations between events on the railway. Clearly, when a train appears on the i-th section (event $ti+$), it means that the same time it leaves the (i-1)-th section (event $t(i\text{-}1)-$).

In fact, because the train has a non-zero length we shall consider that it first appears on the i-th section and only then the detector $t(i\text{-}1)$ is reset, i.e. the arrival of the train to the i-th section is a complex event $ti+, t(i\text{-}1)-$.

The train passes onto the (i+1)-th section if it has arrived to the i-th section (complex event $ti+, t(i\text{-}1)-$ has happened) and the semaphore $s(i+1)$ is open.

From the safety traffic requirements follows that the semaphore on the i-th section ($si+$) can be opened only if the train has left the (i+1)-th section (event $t(i+1)$ and $s(i+2)-$) and the semaphore must be closed as soon as the train appears on the i-th section ($ti+$).

These casual dependencies are graphically shown in Figure 7,a. Based on them, the STGs for the railway system with 6 and 7 sections are generated (see Figure 7,b,c). In the next sections we will discuss how to get correct control circuits from these STG.

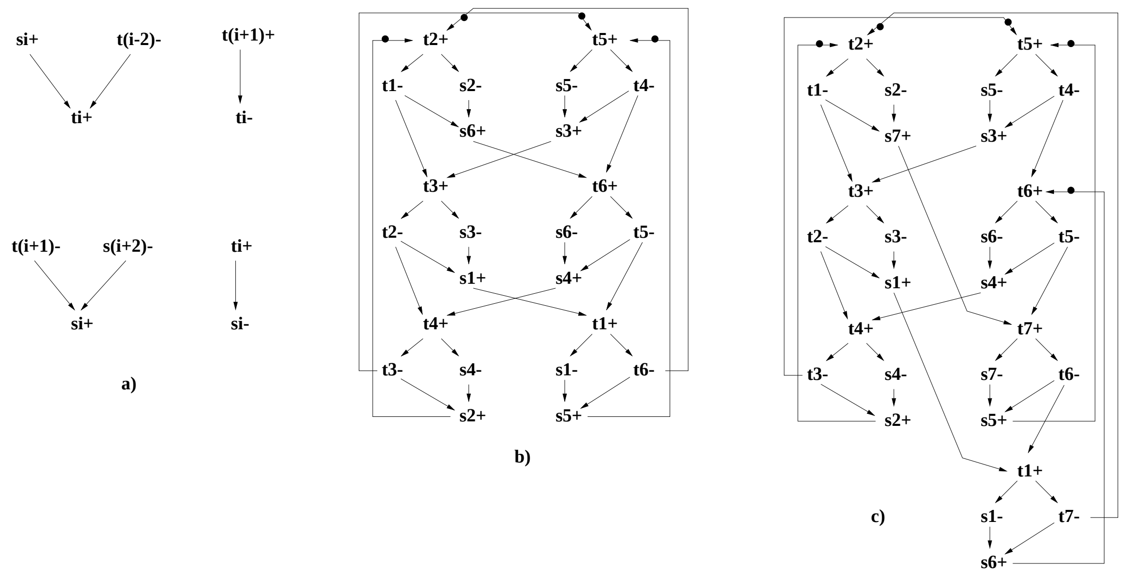


Figure 7: Behavior specification of the "safe" traffic railway. The initial marking indicates that trains are initially in sections 1 and 4.

Dining philosophers. This is the well-known problem [18] of sharing resources (forks) between concurrent processes (philosophers). The system contains n forks for eating spaghetti and n philosophers sitting around the table (see Figure 8,a for the system from 3 philosophers). To eat spaghetti, a philosopher needs two forks[3]: one from the left and one from the right. Clearly, when a philosopher i is eating, his left and right neighbor are missing the fork and they should wait until a philosopher i finishes and frees the forks. Taking left and right forks for a philosopher are independent actions, e.g., he can take one fork and then wait until the other is free. However,

[3]Let us ignore, for the sake of simplicity, whether Italian table manners require this eating protocol.

after eating, a philosopher returns forks simultaneously. Let us refine this classical formulation and allow the philosopher to return forks independently. This gives additional flexibility in describing a philosopher's behavior. For example, instead of the standard "wise and fair" philosopher, we can now describe a "malicious" philosopher who can intentionally delay returning one of the forks to annoy his neighbor. In Section 7 we will see that this complication has crucial consequences for the complexity of verification.

The specification of each philosopher's behavior is as follows. When a philosopher i is ready to eat, he produces the request for eating (eat_i+) and takes the left (lf_i+) and the right forks (rf_i+) if they are available (places F_i and F_{i+1} respectively contain token). After eating (eat_i-), a philosopher returns forks (lf_i- and rf_i-) and returns to the initial state. The specification for three philosophers is presented by the STG in Figure 8,b. It is easily scalable for n philosophers.

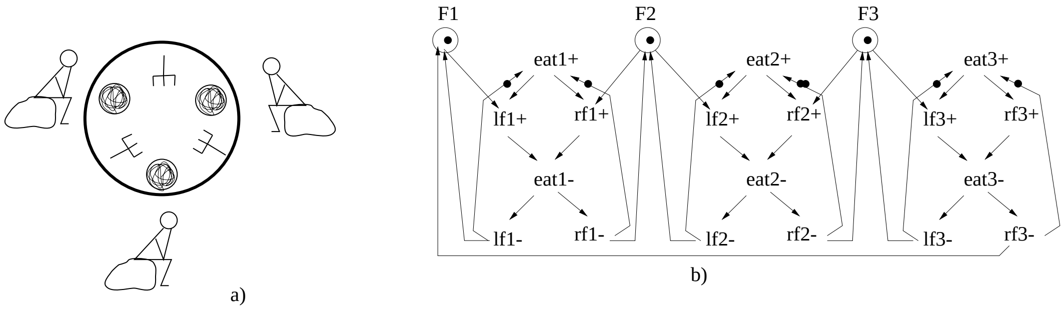


Figure 8: Three dining philosophers

4.1 Circuit Petri Nets

In the previous examples PNs were used as a behavioral specification of a system to be designed. Another possible application arises when one would like to extract a behavior from a system that has already been designed and is specified by a netlist of gates (or by logic equations). A set of logic equations can be syntactically translated into a Petri Net [58]. This PN (though looking a bit awkward) can then be translated into a Reachability Graph and a corresponding SG via a token flow simulation. From the SG a new circuit can be synthesized. This suggests an efficient way for resynthesis and optimization.

The behavior of each circuit signal a can be represented by a so-called *signal cycle* (see Figure 9,a). Places $a0$ and $a1$ of a signal cycle denote the markings in which signal a has value 0 and 1 respectively. Transition $a+$ ($a-$) should fire if signal a is at 0 (1) and the logic function for signal a is at the opposite value 1 (0).

Let us show how to construct circuit PNs using a logic equation of signal a ($a = F(a, b, c, \ldots)$). The *cofactor* of Boolean function F with respect to a ($\overline{a}$) is defined as $F_a = F(a{=}1, b, c, \ldots,)$ ($F_{\overline{a}} = F(a{=}0, b, c, \ldots,)$).

Clearly, all states in which $a{=}0$ and the logic function F equals to 1 are specified by $F_{\overline{a}} = 1$. Hence transition $a+$ in a signal cycle must have input places that should express the condition $F_{\overline{a}} = 1$. Transition firing in PNs has AND-semantics. Therefore, in order to express the condition $F_{\overline{a}} = 1$ via input places of transition $a+$ ($\bullet a+ = \{p_1, p_2, \ldots\}$) the Boolean function $F_{\overline{a}}$ must be represented as $p_1 \wedge p_2 \wedge \ldots$. For this, we will use the *conjunctive normal form* of $F_{\overline{a}}$ [4].

[4] Modeling based on disjunctive normal forms is possible as well. It allows one to model any circuit with a safe PN [58].

The idea of this construction is illustrated by Figure 9,b. Let signal a be specified by the logic equation $a = b\overline{c} + \overline{b}c + ab$. The cofactor of the Boolean function for a with respect to $\overline{a}$ is $F_{\overline{a}} = b\overline{c} + \overline{b}c$ or in a conjunctive normal form $F_{\overline{a}} = (b+c)(\overline{b}+\overline{c})$. Place $p1$ models the condition $b + c = 1$: tokens arrive in $p1$ by the firing of transitions $b+$ and $c+$ and are removed by $b-$ and $c-$. Similarly, place $p2$ models the condition $\overline{b}+\overline{c} = 1$. When both $p1$ and $p2$ are marked, function $F_{\overline{a}}$ equals to 1 and transition $a+$ might fire. Clearly, the change of signal a from 0 to 1 does not influence the value of $F_{\overline{a}}$, and hence places $p1$ and $p2$ are connected with $a+$ by self-loops (A $p \leftrightarrow t$ arc denotes two arcs: $p \rightarrow t$ and $p \leftarrow t$).

The condition for the firing of transition $a-$ is given in the same way by a conjunctive normal form of $\overline{F_a}$.

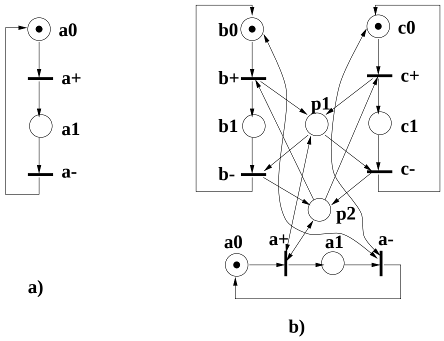


Figure 9: Modeling the circuit behavior by PN

C-latch. A C-latch is an asynchronous latch whose output c takes the value 1 when both inputs a and b are in 1 and takes the value 0 when $a = b = 0$ (c keeps the value when $a \neq b$). The next-state function of a C-latch with inputs a, b and output c is $c = ab + c(a+b)$. As an example of a bottom-up approach in design let us consider the circuit in Figure 10,a which presents a two-input C-latch with output c closed on latch inputs via inverters a and b. The corresponding set of logic equations is shown in Figure 10,b. The Boolean function for the output c of a latch gives two cofactors: $F_{\overline{c}} = ab$ and $F_c = a + b$. Hence transition $c+$ fires when $c = 0$ (place $c0$ is marked) and $a = b = 1$ (places $a1$ and $b1$ are marked), and $c-$ fires when $c = 1$ and $a = b = 0$ ($\overline{F_c} = \overline{a}\overline{b}$). These conditions do not require to introduce additional places in a circuit PN because they can be realized directly on transitions $c+$ and $c-$. The circuit PN corresponding to a C-latch with two inverters is shown in Figure 10,c. After removing redundant arcs we can obtain the STG in Figure 10,d. By observing the behavior of this STG it is easy to see that a C-latch performs the synchronization of two parallel processes on its inputs.

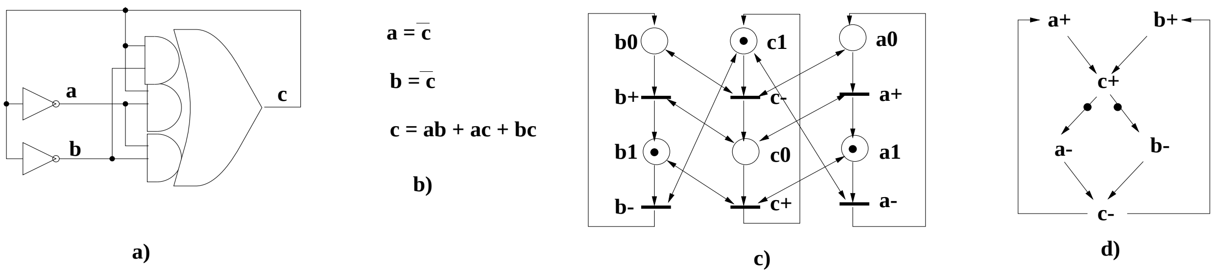


Figure 10: Modeling the C-latch behavior by a circuit PN

5 Implementing PNs with asynchronous circuits

5.1 Speed-independent circuits

A circuit is an interconnection of logic gates. Every gate is composed of (1) an instantaneous logic function evaluator and (2) a delay attached to the function output (see Figure 11). All the delays involved in the switching and transmission of signals within a gate and along the connecting wire prior to its fork are assumed to be reduced to the output delay. The skew of the signals after the fork due to wire delays is assumed to be less than the minimum gate delay. A delay induced by a wire may be modeled explicitly, if needed, by inserting an auxiliary component, a buffer, into the wire break.

For an asynchronous behavior the exact value and even bounds for the delays are often unknown, hence we will be rather pessimistic and assume that in a circuit the delays of gates are finite but unbounded. A circuit whose behavior is correct under any values of gate delays is called *speed-independent* [36]. In this paper we will consider only the design of speed-independent circuits [5].

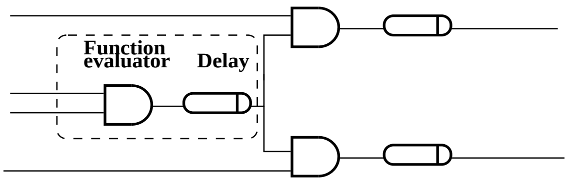


Figure 11: Model of a circuit gate

A *state* of a circuit is a binary vector of the signal values on the outputs of gates and on the input nodes. The behavior of the ith gate can be described by the Boolean equation $z_i' = F_i(A_i)$, where A_i are the values of the signals in the nodes corresponding to the inputs of the ith gate, z_i' is its next (after some delay) value, and F_i is the logic function of the ith gate.

A signal, z_i, is *stable* in state s, if its value in this state, 0 or 1, is equal to the value computed by its logic function, $z_i(s) = F_i(s)$. Otherwise, a signal is said to be *enabled.* For example, an AND gate with output at 1 and at least one input at 0 is *enabled*. In any state an enabled signal can change its value. This generates the dynamic behavior of a circuit, i.e. transitions from the initial state to other states.

The key issue of correctness of an asynchronous circuit operation is *hazard-freedom.* A *hazard* appears as a pulse (possibly a short spike), which does not correspond to any signal transitions in the specification.

An example of hazardous behavior is shown in Figure 12. The circuit in Figure 12,b implements the output c as the XOR function of inputs a and b. If inputs make a transition from $a = 1, b = 0$ to $a = 0, b = 1$, then according to the specification the output c must stay at 1. However, in the circuit it can deviate to 0 – a hazard may occur.

The reason for hazards in an asynchronous circuit is that the signals propagate from inputs to outputs with different delays and can compete with each other. Consider, for example, the circuit

[5]Note that other design styles for asynchronous circuits using PN are also possible, see e.g. methods for synthesis of bounded delay circuits in [28].

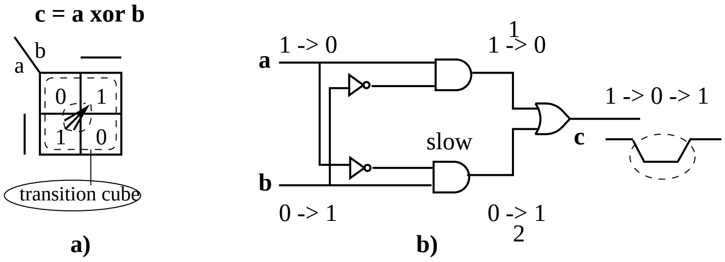


Figure 12: Hazards in a logic circuit

in Figure 12,b. After the output of the upper AND gate has changed from 1 to 0, the output of the OR gate becomes enabled to change from 1 to 0 as well. However, depending on the relative delays of the gates in the circuit, it may well be that the output of the bottom AND gate changes first from 0 to 1. This race at the inputs of the OR gate can lead to a short spike at its output.

Speed-independent circuits, being correct under any distribution of delays, are free from hazards. Indeed, in a speed-independent circuit any enabled signal cannot be disabled before changing its value. Indeed, a hazard as in the previous example can occur only because an output becomes disabled due to changes of the gate inputs, since in that case the gate output might or might not switch depending on the gate delay. This property of "proper disabling" is characteristic of speed-independent circuits.

5.2 Implementability

In order to analyze the implementability of an STG by a circuit, we must introduce a notion of equivalence between PNs (in turn inducing an equivalence between STGs) and a notion of equivalence between circuits and STGs. These notions of equivalence may vary depending on the required level of detail (e.g., whether internal signals in a circuit matter or not). The behavior of PNs can be compared by comparing the languages generated by the PNs, where the language of a PN is defined (as usual) as the *set of its feasible traces*.

Let us start by informally establishing two forms of equivalence of PNs:

- **Strong Equivalence.** Two PNs are strongly equivalent if (1) the number of transitions is the same for both nets and (2) their languages (sets of feasible traces) coincide up to transition renaming.

- **Trace Equivalence.** The set of transitions for each PN is first partitioned into observable (external) and non-observable (internal) transitions. Two PNs are trace equivalent if their languages coincide up to renaming observable transitions if all non-observable transitions are removed from the feasible traces [47].

Contrary to strong equivalence, trace equivalence allows one to map several transitions of one PN into one transition of the other and to ignore internal transitions, checking the equivalence only by observable external transitions. The internal behavior of two PNs can be quite different (e.g., in the number of firing transitions) as long as the external behavior stays the same.

Similarly, one can talk about the equivalence of an STG and its circuit implementation by comparing the respective languages[6]. Such comparison is possible since both STGs and circuits

[6]A weaker notion of conformance can be defined as in [11] for hierarchical verification. However, this topic is outside the scope of this review.

define feasible traces of signal transitions (that can be, e.g., represented using state graphs). We say that an STG, D, is

- *gate-implementable* if there is a logic circuit C strongly equivalent to D
- *implementable* if there is a logic circuit C trace equivalent to D.

The gate-implementability condition guarantees that the circuit has the same number of signals as the STG, i.e. no additional internal signals are required for implementing the specification. Each signal in the circuit is associated with a Boolean function that can be implemented as a complex gate (hence the word "gate" in the definition). In practice, these complex gates are often too large to be physically implementable as a single cell of the gate library and should be decomposed. This design step is called *technology mapping* and it must be performed without introducing hazards into the circuit, and was considered in [25].

It is often the case that the circuit implementing an STG has more gates than the number of signals in the initial STG specification. There are two reasons for this.

- The initial STG specification often defines a behavior that can be implemented as a set of logic gates only by adding state signals, as explained in Section 5.3.
- The logic decomposition of complex gates into smaller ones also introduces internal signal nodes into the circuit.

The implementability condition captures equivalence of an STG and a circuit up to observable external behavior.

We add the prefix SI- and say that an STG is *SI-implementable (SI-gate implementable)* if the circuit implementing the STG is speed-independent (and hence hazard-free).

5.3 Properties required for implementability

In this section we will discuss the following properties of an STG specification, that are related to its implementability: redundancy, boundedness, consistency, determinism, signal and transition persistency.

Redundancy, Boundedness and Determinism

A place is *redundant* if its removal does not change the language of a PN (i.e. the set of feasible traces). Redundant places do not contribute to any trace behavior of a PN and therefore can be removed without altering the specification. For the sake of simplicity, from now on we will consider only irredundant PNs and STGs.

Any irredundant specification that can be implemented by a logic circuit must have a finite state space [7]. Hence the *boundedness* of the underlying irredundant STG is a necessary condition for implementability.

An STG (and the corresponding SG) is called *deterministic* with respect to signal transition $a*$ if for any state s there is at most one state $s1$ such that $s \xrightarrow{a*} s1$. The SG (and corresponding STG) is deterministic if it is deterministic for all signal transitions. Since we consider here only deterministic circuits, determinism of the specification is a necessary condition for the implementability.

[7] A redundant STG specification can produce an infinite marking space and still be implementable. A trivial example can be obtained by inserting into the STG an additional place p, such that $\bullet p = t$, where t is a live transition and $p\bullet = \emptyset$. Place p is unbounded in the new STG but does not affect the behavior of any transition.

Consistency

Not every bounded STG can be interpreted as the specification of the switchings of a set of circuit gates. Let us assume, for example, that the following sequence is feasible in an STG: $b_1+, a+, b_2+, \ldots$. After firing b_1+ signal b must be at logical 1, and no correct interpretation can be suggested for the following transition b_2+. The property of *consistency of state assignment* guarantees that such problem never occurs.

An STG and the corresponding SG is *consistent*[8] if for any feasible trace from the initial state, rising and falling transitions alternate for each signal. All specifications considered so far in this paper have been consistent.

If an STG is bounded, consistent and deterministic, then it can be implemented with a circuit. However, additional state signals may be required to resolve state conflicts.

Complete State Coding (CSC)

Let us illustrate, by the example of the consumer-producer system, why inserting state signals can be necessary. The binary encoded SG corresponding to the STG specifying the system with two consumers is shown in Figure 5,b. There are two states encoded with the same binary code 0001 but with a different next value for the output signal *prod*: in one state, signal *prod* is stable high, while in the other state it is enabled to fall. Hence the definition of the next state function for signal *prod* would require two contradictory values 1 (due to the first state) and 0 (due to the second state) for the same binary vector 0001. Therefore, no circuit with two Boolean functions (corresponding to output signals *con* and *prod*) can implement the STG from Figure 5,a.

A pair of states in an SG with the same binary encoding but different enabling of non-input signals is said to be in a *Complete State Coding (CSC)* conflict. An STG and the corresponding SG satisfy the *Complete State Coding* property if no pair of SG states has the same encoding but different enablings of non-input signals.

If a bounded and consistent STG with n non-input signals has the CSC property, then it can be implemented by a circuit with n (arbitrarily complex) logic gates.

CSC conflicts can be resolved by encoding the corresponding states with additional signals. These signals distinguish the CSC conflicts by assigning the corresponding states different binary codes. In [5] an effective procedure for inserting additional signals to resolve CSC conflicts is presented. This procedure preserves the trace equivalence and speed-independence of the original specification and always converges for safe STGs.

Persistency

All the properties considered up to now do not restrict hazards in the implementation. To capture the speed-independence requirement at the STG level, let us consider the *signal persistency* property[9].

Signal persistency (similarly to speed-independence, as defined above for circuits) means that if an STG signal is enabled, *it fires independently from the firing of other signals*[10]. However, one should distinguish between input and non-input signals. For inputs, which are controlled by the environment, it is possible to have a non-deterministic choice which is represented in the STG

[8] Sometimes we say that an STG or an SG has a consistent state assignment.

[9] This notion of persistency is not related to that used by [4]. The latter, as shown in [23], is loosely connected with the number of cubes in a sum-of-products implementation of each signal.

[10] In this paper we do not consider STG specifications, that can be implemented with non-deterministic circuits like arbiters. See [7] for a detailed discussion.

model by conflicts, i.e., disabling of one input signal by another input signal. Input conflicts *per se* do not imply hazardous behavior. For non-input signals, which are produced by circuit gates, signal transition disabling may lead to hazards at the output of the gate, as discussed above, thus making the circuit behavior dependent on the gate delays. Disabling input transitions by non-input transitions, on the other hand, is also considered dangerous, since it can lead to hazards in a logic implementation of the environment[11].

Definition 5.1 *An* SG *(and the corresponding* STG*) is* signal persistent *if:*

1. *no* non-input *signal can be disabled by another signal and*
2. *no* input *signal can be disabled by a non-input signal.*

A bounded, consistent, deterministic and signal persistent STG can be implemented with a speed-independent circuit.

Signal persistency and transition persistency (see Section 2) are closely related. Clearly, the only source of signal non-persistency of a signal a is the non-persistency of some transition labeled with a_i*. Yet, not any non-persistency of a_i* leads to the violation of persistency by signal a. In Figure 13,a transitions labeled with a_1+ and b_2+ are both non-persistent. However, signals a and b are persistent in the corresponding SG in Figure 13,c.

Although firing of a_1+ disables b_2+, it also enables transition b_1+. So, both before and after the firing of a_1+, signal b remains enabled. By the sets of feasible traces such a behavior of signals a and b is equivalent to the concurrent firing of $a+$ and $b+$[28]. Therefore, both STG $D1$ and $D2$ have the same SG[12] (Figure 13,c). One can conclude that for signal b the conflict of the transition b_2+ is "fake".

For fake-free STGs signal persistency and transition persistency coincide, and hence signal persistency can be checked as transition persistency at the STG level without generating a state graph. It can also be shown [24] that fake conflicts for non-input signals can be always replaced by concurrency in STG specifications, and hence excluding fake conflicts does not restrict expressiveness of STG specifications.

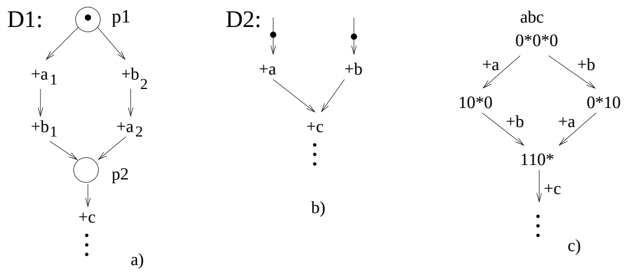


Figure 13: Transition and signal non-persistency

In summary we can conclude that a bounded, consistent, and deterministic STG is

- always implementable by a logic circuit;

[11] A logic implementation of the environment would also be hazardous in the presence of input-to-input disabling. However, this is considered acceptable here because this non-determinism is often the result of an *abstraction* in the environment behavior specification, for the purpose of simplifying the STG model. An environment in general must have a persistent specification in order to be implementable as a hazard-free logic circuit.

[12] They can be distinguished, if necessary, if a true concurrency semantics is used instead of interleaving.

- gate-implementable, if it satisfies the CSC requirement;
- SI-implementable, if it is signal persistent.

6 State-based verification

The properties of STG implementability are formulated in Section 5.3 in terms of SG states. However, the explicit manipulation of an SG is often problematic because of the exponential growth of its size with respect to the number of signals. Recent developments in using *symbolic* techniques for reachable state space traversal based on Binary Decision Diagrams(BDDs) [2, 42], can be applied to avoid the explicit manipulation with the state set and to soften the space explosion problems.

6.1 Modeling bounded Petri nets and STGs using Boolean functions

Let $N = \langle P, T, F, m_0 \rangle$ be a safe Petri net and let M_P be the set of all markings of N ($n = |P|, |M_P| = 2^n$). A marking can be represented by a Boolean vector $m = (p_1, \ldots, p_n)$, where $p_i = 1$ ($p_i = 0$) denotes that p_i is marked (not marked)[13]. Each set of markings $M \in 2^{M_P}$ can be represented by a characteristic logic function $\chi_M : B^n \to B$ that equals 1 for those vectors that correspond to markings in M. For example, given the Petri net depicted in Figure 5,a, the characteristic function of the set of markings $M = \{$(0,1,0,0,1,0,0,0,0), (0,1,1,0,1,0,0,0,0), (1,1,0,0,1,0,0,0,0), (1,1,1,0,1,0,0,0,0), (1,1,1,1,1,0,0,0,0) $\}$ is calculated as the disjunction of Boolean vectors $m \in M$. The resulting function is $\chi_M = p_1 p_4 \overline{p_5}\,\overline{p_6}\,\overline{p_7}\,\overline{p_8}(p_0 p_2 + \overline{p_3})$.

The *transition function* of a Petri net is a function $\delta_N : 2^{M_P} \times T \longrightarrow 2^{M_P}$ that transforms, for each transition, a set of markings M_1 into a new set of markings M_2 as follows: $M_2 = \delta_N(M_1, t) = \{m_2 \in M_P : \exists m_1 \in M_1, m_1 \xrightarrow{t} m_2\}$. Given a set of markings M_1 the transition function calculates a new set of markings M_2, reachable in one step from M_1 Computation of the transition function can be efficiently implemented by using the structural information of the PN.

Before we give an expression for the transition function let us introduce the characteristic functions of some important sets related to a transition $t \in T$:

$$
\begin{aligned}
\mathrm{E}(t) &= \bigwedge_{p_i \in {}^\bullet t} p_i \quad (t \text{ enabled}),\\
\mathrm{ASM}(t) &= \bigwedge_{p_i \in t^\bullet} p_i \quad (\text{all successors marked}),\\
\mathrm{NPM}(t) &= \bigwedge_{p_i \in {}^\bullet t} \overline{p_i} \quad (\text{no predecessor marked}),\\
\mathrm{NSM}(t) &= \bigwedge_{p_i \in t^\bullet} \overline{p_i} \quad (\text{no successor marked}).
\end{aligned}
$$

Function $E(t)$ ($ASM(t)$) states that all input (output) places of transition t contain a token, and function $NPM(t)$ ($NSM(t)$) states that no input (output) place of t contains a token.

Let us recall a few useful definitions from the theory of Boolean functions. Let $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ be a Boolean function of n variables. The cofactor of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to literal x_i is $f(x_1, x_2, \ldots, 1, \ldots, x_n)$, while the cofactor with respect to literal x_i' is $f(x_1, x_2, \ldots, 0, \ldots, x_n)$. The notion of cofactor can be generalized to a set of variables, e.g., $f_{x_i, x_2} = (f_{x_1})_{x_2}$. The existential abstraction of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to x_i is $\exists_{x_i}(f) = f_{x_i} + f_{x_i'}$.

[13] The restriction of safeness is considered only for clarity. Unsafe but bounded places can be represented in the same framework by several Boolean variables [42].

We can now express the set of markings $\delta_N(M,t)$ that are reachable from a set of markings M by considering only one transition t. A marking m is in $\delta_N(M,t)$

- either if $m' \in M$ and $m' \xrightarrow{t} m$,
- or if $m \in M$ and t is not enabled in m.

The symbolic representation is as follows:

$$\delta_N(M,t) = (M_{\mathrm{E}(t)} \cdot \mathrm{NPM}(t))_{\mathrm{NSM}(t)} \cdot \mathrm{ASM}(t),$$

The existential quantification of t from the above formula gives the set of markings $\delta_N(M)$ reachable by firing any *one* enabled transition from a marking in M. Here F_g denotes the cofactor of Boolean function F with respect to the set (Boolean product) of literals g.

Example. Assume that in the example of Figure 5 we calculate $M_1 = \delta_N(M, req1+)$ given the set $M = p_0 p_1 \overline{p_2}(p_5\overline{p_6} + \overline{p_5}p_6) + \overline{p_1}p_3p_5\overline{p_6}p_7$. First, $M_{\mathrm{E}(req1+)}$ (cofactor of M with respect to $\mathrm{E}(req1+) = p_1$) selects those markings in which $req1+$ is enabled and removes its predecessor places from the characteristic function ($M_{\mathrm{E}(req1+)} = p_0\overline{p_2}(p_5\overline{p_6} + \overline{p_5}p_6)$). Then the product with $NPM(req1+) = \overline{p_1}$ eliminates the tokens from the predecessor places ($M_{\mathrm{E}(req1+)} \cdot \mathrm{NPM}(req1+) = p_0\overline{p}_1\ \overline{p_2}(p_5\overline{p_6} + \overline{p_5}p_6)$). Next, the cofactor with respect to $\mathrm{NSM}(req1+) = \overline{p_2}$ removes all the successor places obtaining $(M_{\mathrm{E}(req1+)} \cdot \mathrm{NPM}(req1+))_{\mathrm{NSM}(req1+)} = p_0\overline{p_1}(p_5\overline{p_6} + \overline{p_5}p_6)$. Finally, the product with $ASM(req1+) = p_2$ adds a token in all the successor places of $req1+$ ($M_1 = p_0\overline{p_1}p_2(p_5\overline{p_6} + \overline{p_5}p_6)$).

Let $D = \langle N, S_A, \lambda\rangle$ be an STG with N as underlying Petri net. Let G be the SG corresponding to D, and let S be the set of binary states of G. Since there is a correspondence between markings of N and states of G, we represent the full state of the STG by the vector $y = (m, s)$, where m is a marking of N and s the state code of the corresponding state in G, respectively.

The transition function can now be extended for STGs as a function $\delta_D : 2^{(M_P \times S)} \times T \longrightarrow 2^{(M_P \times S)}$. For a set of full states M_F, δ_D is defined as follows:

$$\delta_D(M_F, t) = \begin{cases} (\delta_N(M_F,t))_{\overline{a}} \cdot a & \text{if } \lambda(t) = a_i+ \\ (\delta_N(M_F,t))_{a} \cdot \overline{a} & \text{if } \lambda(t) = a_i- \end{cases}$$

6.2 Verifying SI-implementability via symbolic traversal

STG implementability properties can be verified by calculating all reachable markings (states) of the STG. Given the initial marking m_0 of N and the initial values of the signals s_0, the set of states of an STG can be calculated by using *symbolic traversal* techniques, similar to those used for the verification of finite state machines [8].

Figure 14 describes an algorithm for symbolic traversal. It starts from an initial full state (m_0, s_0). For each outermost iteration, all transitions of the Petri net are visited and fired from all the new states found so far. The algorithm halts when a fixed point is reached (no new states are generated).

Safeness and Boundedness. The calculation of the reachability space by symbolic traversal is done under the assumption that the Petri net is safe. This calculation is erroneous if some of the markings is unsafe, since unsafe markings are not representable by encoding each place with one signal.

```
traverse_STG (D) {
    Reached = From = {(m_0, s_0)};
    repeat
        for each t ∈ T do
            To = δ_D( From, t);
            From = From ∪ To;
        endfor
        New = From − Reached;
        Reached = Reached ∪ New;
        From = New;
    until ( New = ∅);
    return Reached; /* The set of reachable states of D */
}
```

Figure 14: Algorithm for symbolic traversal of an STG

However, detecting if some unsafe marking is reachable can be done by identifying a marking m in which a transition t is enabled and some successor place p of t, and not predecessor of t, is already marked. In that situation, after firing transition t, place p will have two tokens. This idea is implemented in the algorithm from Figure 15.

```
is_safe (N) {
    for each t ∈ T do {
        Succ_p = 0;
        Enabled = Reached · E_t;    /* Reached – set of reachable markings of N */
        for each (p_i ∈ t• ∧ p_i ∉ •t) do { Succ_p = Succ_p + p_i }
        if (Enabled · Succ_p ≠ 0) return false;
    }
    return true;
}
```

Figure 15: Algorithm for safeness checking

A k-bounded place $p \in P$ can be represented with a set of Boolean variables, $v_1, \ldots, v_q$ to encode the possible number of tokens. The number of required variables depends on the type of encoding. If an *one-hot* encoding is used, k variables are needed. For example, in a 3-bounded Petri net the number of tokens in place p could be represented by three variables. With a *binary encoding* $\lceil \log_2(k+1) \rceil$ variables would be required (see Tab. 1).

# tokens	one-hot encoding	binary encoding
0	$\overline{v}_3\overline{v}_2\overline{v}_1$	$\overline{v}_2\overline{v}_1$
1	$\overline{v}_3\overline{v}_2 v_1$	$\overline{v}_2 v_1$
2	$\overline{v}_3 v_2\overline{v}_1$	$v_2\overline{v}_1$
3	$v_3\overline{v}_2\overline{v}_1$	$v_2 v_1$

Table 1: Encoding of k-bounded places ($k = 3$)

The worst thing about the traversal of an unsafe STG is that the bound on the number of tokens

```
transition_persistency (N) {
    for each p ∈ P, |p•| > 1 do
        for each t_i ∈ p• do
            Enabled = R(N) · E(t_i);
            for each t_j ∈ p•, t_i ≠ t_j do
                if (δ_N( Enabled, t_j) ∩ ¬E(t_i) ≠ ∅)
                error ("t_i disabled by t_j");
            end for
        end for
    end for
}
```

Figure 16: Algorithms to verify persistency

for each place must be guessed in advance in order to compute the transition function $\delta_N(M,t)$. In case the guessed bound of some place is exceeded, the traversal must be restarted from the beginning, with a new definition of δ. Hence we can state that non-safeness is a problematic property for state-of-the-art BDD-based techniques.

Consistency. Verifying that the STG is consistent can be done during the traversal by checking the consistency of the newly generated states. We first define the following characteristic function:

$$\mathrm{E}(a*) = \bigvee_{t:\lambda(t)=a*} E(t) \qquad\qquad (a* \text{ is enabled})$$

The characteristic function of the states with inconsistent assignment is derived according to the definition of consistency (cf. Section 5.3):

$$\begin{aligned}
\text{Inconsistent}(a+) &= E(a+)\cdot a \quad \text{/* a+ enabled and a=1 */}\\
\text{Inconsistent}(a-) &= E(a-)\cdot \overline{a} \quad \text{/* a- enabled and a=0 */}\\
\text{Inconsistent}(a) &= \text{Inconsistent}(a+) + \text{Inconsistent}(a-)\\
\text{Inconsistent}(D) &= \bigvee_{a\in S_A} \text{Inconsistent}(a)
\end{aligned}$$

Let us call $R(D)$ the set of reachable states (markings and binary codes) of the STG D. D is inconsistent if $R(D) \cap$ Inconsistent$(D) \neq \emptyset$.

Persistency. For a fake-free STG, transition persistency coincides with signal persistency. Hence to analyze the persistency of an STG, it is sufficient to analyze the absence of direct conflicts involving non-input transitions.

An algorithm to check transition persistency is shown in Figure 16. Note that only transitions with some common predecessor place can be in a direct conflict. This is used in the algorithm to reduce the search. Let $R(N)$ be the set of reachable markings of N. The set of markings with t_i enabled is calculated. Next, the set of markings reachable in one step by firing some transition $t_j \neq t_i$ is obtained. If t_i is not enabled in any one of those markings, then t_i is not persistent.

Complete State Coding. The CSC requirement can be checked for each non-input signal by defining the following characteristic functions:

$$\begin{aligned}
\mathrm{ER}(a+) &= \exists_P\,(R(D)\cdot E(a+))\\
\mathrm{ER}(a-) &= \exists_P\,(R(D)\cdot E(a-))\\
\mathrm{QR}(a+) &= \exists_P\,(R(D)\cdot a - E(a-))\\
\mathrm{QR}(a-) &= \exists_P\,(R(D)\cdot \overline{a} - E(a+))\,,
\end{aligned}$$

where $\exists_P F$ denotes the *existential abstraction* of F with respect to the set of all variables representing places of the PN, P. Using this operation removes all variables from P from the corresponding Boolean functions.

$\mathrm{ER}(a*)$ is the set of binary codes that correspond to states in which some a_i* is enabled (*a set of excitation regions*). It is obtained by abstracting the places ($\exists_P$) from the states of the excitation region. $\mathrm{QR}(a+)$ (*a set of quiescent regions*) is the set of binary codes that correspond to states in which $a = 1$, but $a-$ is not enabled (similarly for $\mathrm{QR}(a-)$).

A CSC conflict occurs in an SG when in two states $s1$ and $s2$ with the same binary code some non-input signal a is enabled to rise in $s1$ and is stable at 0 in $s2$, or enabled to fall in $s1$ and stable at 1 in $s2$ This means that state $s1$ belongs to the excitation region of $a*$ while $s2$ belongs to a quiescent region of this signal. As $s1$ and $s2$ have the same binary code, the characteristic functions of excitation and quiescent regions for $a*$ must intersect. Based on this the CSC requirement for non-input signal a can be checked as follows [41]:

$$\mathrm{CSC}(a) = (\mathrm{ER}(a+) \cap \mathrm{QR}(a-) = \emptyset) \wedge (\mathrm{ER}(a-) \cap \mathrm{QR}(a+) = \emptyset)$$

$$\mathrm{CSC}(D) = \bigwedge_{a\ is\ non-input} \mathrm{CSC}(a)$$

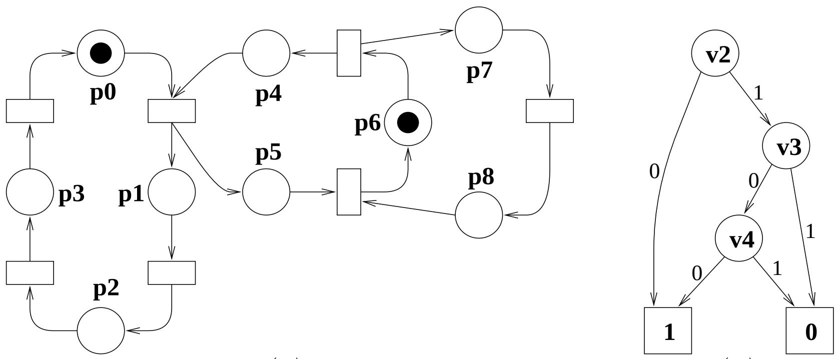
Example. In the example of the consumer-producer system of Figure 5, the set of states where signal *prod* is stable and equal to 1 (quiescent region of *prod*) is $\{0001, 1001, 1011, 0101, 0111, 0011\}$ (see Figure 5,b). The set of states where *prod* is enabled and equal to 1 (excitation region of *prod*$-$) is the single state 0001. The intersection of the excitation region and of the quiescent region is non-empty, thus indicating the presence of a CSC conflict (see Figure 5,b) due to the states with the code 0001.

Determinism and fake conflicts. Determinism and fake conflicts can also be easily checked using manipulations with characteristic Boolean functions.

This Section showed that all the properties related to STG SI-implementability can be checked by symbolic traversal of the reachability space. This is not surprising, because the set of reachable states naturally contains all the information needed for the implementation of an STG. Some effort is still required because in the symbolic traversal the reachability space is manipulated implicitly, and hence the check of some properties can have a complex formulation. However, even with an implicit representation of the reachability set the size of the corresponding BDD might still be exponential with respect to the size of the original STG. For such examples the event-based approach developed in the next section can be more efficient.

6.3 Symbolic representation with Binary Decision Diagrams

In this section we briefly explain how sets of states can be represented by means of Boolean functions and efficiently manipulated by using Binary Decision Diagrams (BDDs) [29].



(a) (b)

place	v_0	v_1	v_2	v_3	v_4
p_0	0	0	-	-	-
p_1	0	1	-	-	-
p_2	1	0	-	-	-
p_3	1	1	-	-	-
p_4	-	-	0	0	-
p_5	-	-	0	1	-
p_6	-	-	1	0	-
p_6, p_7	-	-	-	-	0
p_8	-	-	-	-	1

(c)

Figure 17: (a) Petri net, (b) BDD representing the set of reachable markings and (c) place encoding.

A BDD is a directed acyclic graph with one root and two leaf nodes (0 and 1). Each non-leaf node is labeled with a Boolean variable and has two outgoing arcs with labels 0 and 1. A BDD represents a Boolean function interpreted as follows: each variable assignment has a corresponding path that goes from the root node to one of the leaf nodes. The label of the leaf node is the value of the function for that assignment. As an example, the BDD depicted in Figure 17.b represents the function $f(v_2, v_3, v_4) = \bar{v_2} + \bar{v_3}\bar{v_4}$. The value of the function for the assignment $v_2 = v_3 = 1$ and $v_4 = 0$ is 0. This assignment corresponds to the path $v_2 \xrightarrow{1} v_3 \xrightarrow{1} 0$. We refer the reader to [2] for further details on how to manipulate Boolean functions efficiently by means of BDDs.

Figure 17 illustrates how the reachability set of markings of a PN can be represented with BDDs. The example relies on the fact that an efficient encoding can be found to represent the reachable markings.

The strategy for encoding is based on the observation that the sets of places $SM_1 = \{p_0, p_1, p_2, p_3\}$, $SM_2 = \{p_4, p_5, p_6\}$ and $SM_3 = \{p_6, p_7, p_8\}$ are state machines of the PN [17]. This information can be structurally obtained by using algebraic methods [9]. State machines correspond to place-invariants of the PN and preserve their token count in all reachable markings. Given the initial marking of the net, at most one of the places of each state machine will be marked at each marking. Thus, the following encoding can be proposed: two Boolean variables (v_0 and v_1) can be used to encode the token in SM_1, two Boolean variables (v_2 and v_3) for SM_2. Only one Boolean variable (v_4) is sufficient for SM_3, since SM_2 already encodes p_6 and only p_7 and p_8 must be distinguished.

Figure 17.c proposes an encoding for the places in such a way that the set of reachable markings can be represented by the Boolean function $S = \bar{v_2} + \bar{v_3}\bar{v_4}$. Note that the function has 20 minterms and that each of them corresponds to a reachable marking of the function. Here are some examples of how the markings (represented as sets of marked places) are encoded:

$$
\begin{aligned}
(0,1,0,1,1) &\rightarrow \{p_1, p_5, p_8\} \\
(1,1,1,0,0) &\rightarrow \{p_3, p_6\} \\
(0,0,1,0,1) &\rightarrow \text{unreachable marking}
\end{aligned}
$$

With such representation, the basic operations on sets of states (union, intersection, complement) can be mimicked as Boolean operations on Boolean functions (or, and, not). Moreover, such functions can have a compact representation.

7 Event-based verification. Unfolding approach

Methods based on partial orders are well-known techniques to avoid the "state explosion problem" in the behavioral analysis of Petri Nets (PN) [32, 13, 22]. Instead of a reachability graph, a finite prefix (called unfolding) of an equivalent occurrence net (acyclic net where all places have at most one input transition) is generated. Although we consider the application of the unfolding approach to the same verification tasks as BDD-based methods of Section 6, these two techniques are unfortunately incomparable in terms of efficiency (i.e., it is very difficult to tell a priori which one is better, except for some very special properties and subclasses of PNs). Hence, they should be viewed as complementary rather than completely independent.

7.1 Petri Net unfolding

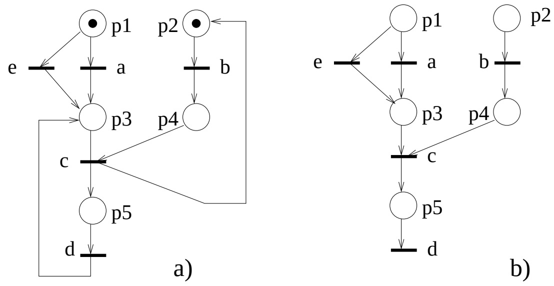


Figure 18: Cyclic (a) and acyclic PNs (b).

Let us call a PN *acyclic* if there are no cycles in the graph of the PN. In an acyclic PN some places have no input transitions. We assume that all these places are initially marked with one token and no other places are initially marked. Two examples of a cyclic and an acyclic PN are shown in Figure 18,a,b.

Definition 7.1 (Ordering relations) *Let $N = \langle P, T, F \rangle$ be an acyclic PN and $x_1, x_2 \in P \cup T$.*

- x_1 precedes x_2 *(denoted by $x_1 \Rightarrow x_2$) if (x_1, x_2) belongs to the reflexive transitive closure of F, i.e., there is a path in the graph of the PN between x_1 and x_2.*

- *x_1 and x_2 are in* conflict *(denoted by $x_1 \# x_2$), if there exist distinct transitions $t_1, t_2 \in T$ such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, and $t_1 \Rightarrow x_1$, and $t_2 \Rightarrow x_2$.*

- *x_1 and x_2 are* concurrent *(denoted by $x_1 || x_2$), if they are neither in precedence, nor in conflict.*

Let us consider these relations for the example of an acyclic PN in Figure 18,b. Directly by Definition 7.1 one can find that $b \Rightarrow c \Rightarrow d$, $p1||p2$, $p1||p4$, $a||b$, $a\#e$.

Acyclic PNs are easier to analyze because their properties can be completely specified by ordering relations [26]. This suggests an effective way to analyze and/or manipulate general PNs: first a cyclic PN is unfolded into an equivalent acyclic net, and then a finite prefix of the acyclic net is used. It is particularly convenient to unfold a PN into a special kind of acyclic PN, called *occurrence net* in [39, 13], in which every place has at most one input transition.

Figure 19,a shows an occurrence net for the PN from Figure 18,a. Each transition ti (place pj) of the initial PN has a set of corresponding transitions (places) in the occurrence net $ti', ti'', ti''', \ldots$ $(pj', pj'', pj''', \ldots)$ that are called *instantiations* of ti (pj). It can be shown that under the partition

r that associates each transition t of PN with its instantiations $t', t'', \ldots$, the original PN is trace equivalent to its occurrence net. Further we will refer to any object in the occurrence net equivalent to an object in the original cyclic PN by adding one or more apostrophes (or by adding a superscript) to its name. For example, t' and t are the corresponding transitions in an occurrence net and a PN, m' is a marking in the occurrence net and m is the corresponding marking in the PN, etc.

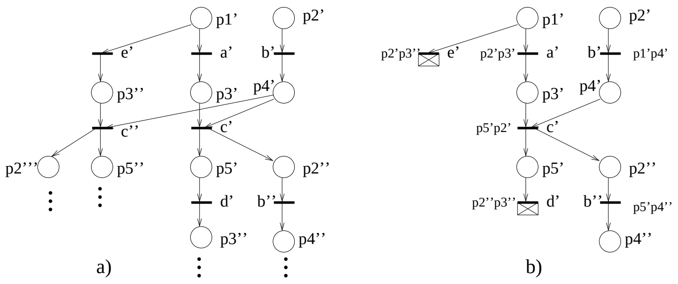


Figure 19: Occurrence net (a) and unfolding (b).

Although the occurrence net for a cyclic PN can be infinite, it is always possible for a bounded PN to truncate the occurrence net up to a finite “complete” subgraph (an *unfolding*) that carries the same amount of information. We need to introduce several notions in order to define this truncation.

Definition 7.2 (Configurations) *[32].*

- *A set of transitions $C' \subset T'$ is a* configuration *in an occurrence net if:*

  1. *for each $t' \in C'$ the configuration C' contains t' together with all its predecessors;*
  2. *C' contains no mutually conflicting transitions.*

- *The minimal configuration that contains t' and all the transitions preceding t' is called a* local configuration *of the transition t' (denoted $\{\Rightarrow t'\}$)*

Each configuration C' corresponds to a marking (called *final marking* of C') that is reachable from m_0 after all the transitions in C' have been fired. A final marking of a local configuration of t' is called a *basic marking* of t' and denoted $BM(t')'$.

In the occurrence net of the PN from Fig. 19,a, the local configuration $\{\Rightarrow c'\}$ for transition c' is equal to $\{a', b', c'\}$. The basic marking for c' is $BM(c')' = \{p5', p2''\}$. Marking $\{p3', p4'\}$ does not correspond to any local configuration; however, it corresponds to the configuration $\{a', b'\}$.

7.2 Cut-offs

Occurrence nets are truncated by *the cut-off* transitions. Criteria for choosing these transitions for different classes of PNs were suggested in [32, 26, 20]. The necessary condition for a transition t_i' to be a cut-off is that the basic marking of t_i' repeats the basic marking of some other transition t_j' that has been generated earlier in an occurrence net (t_j' is called *an image* of t_i').

Intuitively it means that firing t_i generates a marking of the original PN that has has already been reached after firing t_j. As shown in [32, 26, 20] some more precautions have to be taken in determining cut-offs.

Definition 7.3 *[20] A transition t'_i of the occurrence net is a* cut-off *transition, if there exists another transition t'_j such that*

1. $BM(t'_i) = BM(t'_j)$ *and*

2. $\{\Rightarrow t1\} \prec \{\Rightarrow t2\}$,
where $\prec$ is an adequate partial order.

The simplest partial order is defined as the integer ordering on the size (defined as the number of transitions) of configurations [32]. More elaborate (and more efficient) cut-off criteria have been defined in [26, 20]. An *unfolding* is obtained from the occurrence net by removing all the places and transitions which follow cut-offs.

The unfolding corresponding the PN of Figure 18,a is shown in Figure 19,b. The basic markings are pointed near the transitions. It is easy to see that transitions e', d' and a' have the same basic marking $p2p3$ in the PN, and hence e' and d' are the cut-offs with the same image a'. The parts of the occurrence net (see Figure 19,a) "after" cut-offs is absent in the unfolding.

It was shown in [32] that no reachable marking of the original PN is lost in the unfolding. An "adequate partial order" is needed for metrics on a set of configurations. For different subclasses of PNs, different partial orders were proposed (see a diagram in Figure 20):

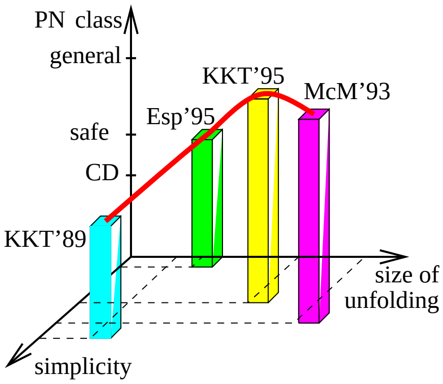


Figure 20: Different cut-off criteria

- Number of transition instantiations for Change Diagrams[14] (KKT'89) [22]
- Size of configurations (number of transition in configuration) (McM'93) [32]
- Comparison of Parikh vectors [9] for the Foata-Normal-Form [10] of configurations for safe PNs (Esp'95) [20]
- Enhanced cut-offs (KKT'95) [26]

The last two cut-off criteria are the most general and efficient, in terms of the size of the generated unfolding. The criterion of Esp'95 [20] is optimal, in the sense that it allows one to define a total order between transitions with the same basic marking. I.e., each time two transitions with the same basic marking are generated in an unfolding, one of them will be defined as a cut-off. The unfoldings obtained by this method have a minimal size. The shortcoming of the

[14]Roughly speaking, a CD models only AND-causality, OR-causality and concurrency, but no conflicts. In practice, it is roughly comparable to a Marked Graph.

approach [20] is the restriction to the class of safe PNs [15]. Enhanced cut-offs [26] work for general PNs, even though they cannot guarantee minimal size unfoldings. The size of the unfolding obtained with the enhanced cut-off criterion is typically much smaller than the size of the RG for a PN. For example, for the 60-users distributed mutual exclusion (DME) arbiter the RG contains approximately 7.0×10^{19} markings, while the unfolding contains only 240 transitions [26].

7.3 Checking implementability properties by unfoldings

Boundedness. Figure 21,a shows an example of an unbounded PN. The place $p3$ is unbounded because the trace $t1, t2, t1, t2, \ldots$ can generate an unbounded number of tokens in $p3$.

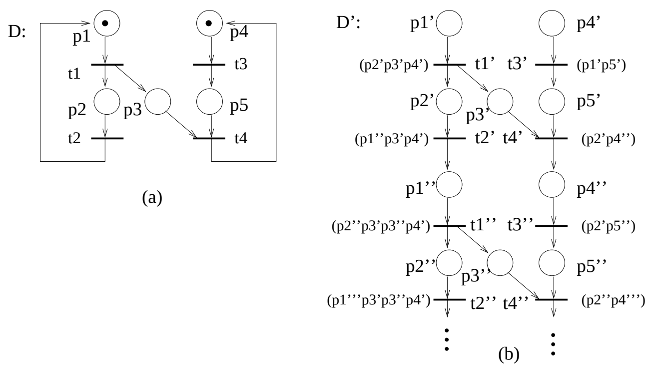


Figure 21: Unbounded PN (a) and its occurrence net (b).

Let N' be an unfolding of PN N. *N is unbounded if and only if there is a transition $t \in T$ that has two instantiations in N', t' and t'', such that $t' \Rightarrow t''$ and $BM(t') < BM(t'')$.*

Hence, checking boundedness of a PN can be reduced to the analysis of the precedence relation between transitions in an unfolding.

Return to the example of Figure 21. As soon as the transition $t1''$ will be generated in the unfolding, one can conclude that the original PN is unbounded. Indeed, both $t1'$ and $t1''$ are instantiations of the same transition $t1$, and $t1' \Rightarrow t1''$. $BM(t1'') = p2p3p3p4 > BM(t1') = p2p3p4$, so the marking of place $p3$ can grow indefinitely and this place is unbounded.

Persistency. An unfolding gives an explicit representation of conflicts. However, not every pair of conflicting transitions is non-persistent (in direct conflict). For example, conflicting transitions $t3', t4'$ from PN in Figure 22 do not share any input place and thus cannot disable each other. Conflicting transitions $t1', t4'$ are also not in a direct conflict (although they share the input place $p2'$), because they are never enabled simultaneously.

The structural properties of direct conflicts can be stated as follows [26]:

A direct conflict occurs when two transitions share the same input place and none of their predecessors are in conflict (otherwise a direct conflict has already occurred for the predecessors).

Hence the transition persistency check can be reduced to the analysis of conflict relations between transitions in an unfolding.

[15] The extension of this method for unsafe nets has been proposed, but the experimental results are not yet known [14].

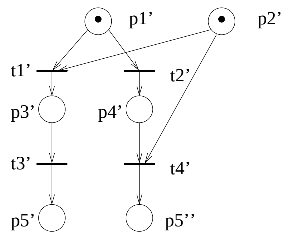


Figure 22: PN with direct and non-direct conflicts

For example, in the PN in Figure 22, transitions $t1'$ and $t4'$ are not in a direct conflict, because $t2'$ is the direct predecessor of $t4'$ and $t2'$ is in conflict with $t1'$. However, $\{t1', t2'\}$ are in a direct conflict.

An STG is non-deterministic whenever two transitions with the same label are in a direct conflict. This is clearly a particular case of non-persistency. Similarly, *fake conflicts* in an STG can be checked based on the direct conflict relation.

Consistency. The consistency property states that in each firing sequence of an STG, rising and falling transitions of the same signal must alternate. To analyze it by unfoldings, we need to formulate consistency in terms of ordering relations between events.

Definition 7.4 *1. An unfolding of an* STG *is called auto-concurrent if it contains two concurrent transitions t_1 and t_2 labeled by the same signal.*

2. An unfolding of an STG *is* sign alternating *if for any pair of rising signal transitions $a_i+ \Rightarrow a_j+$ there exists a falling transition a_k- such that $a_i+ \Rightarrow a_k- \Rightarrow a_j+$. Similarly for falling transitions.*

In the STG shown in Figure 23,a transitions a_1+ and a_2+ are concurrent. The following trace is feasible: $a_1+, a_2+, a-, \ldots$. After firing a_1+ signal a is at logical 1, and therefore the next transition of signal a should be negative. However, the trace contains a positive transition a_2+. This shows that auto-concurrency indeed captures one of the features of consistency.

Example. To illustrate the importance of auto-concurrency and sign alternation, let us return to the railway example with seven rail sections from Section 4 (part of its unfolding is presented in Figure 24,a). One can notice that events $si'-$ and $si'+$ are concurrent in the unfolding (see e.g. $s2'-$ and $s2'+$ in Figure 24,a). This is not the case for the railway system with 6 sections. The reason for such difference in the behavior is that for six sections the traffic of trains is rather strictly synchronized, and each train can move relatively to the other only by one section. With more sections in the system, the freedom of motion increases and our seemingly good specification shows inconsistencies, and hence cannot be implemented. To eliminate this problem, let us change the specification and allow a train to appear on the i-th section strictly after the semaphore on the previous section is closed ($s(i\text{-}1) \rightarrow ti+$). The obtained STG (shown in Figure 24,b) is consistent and can be implemented with a speed-independent circuit. Note that further complication of the system (introducing more sections) does not lead to any incorrectness and the corresponding STG keeps the implementability properties.

Auto-concurrency and sign alternation allow us to check consistency for all but the "last" signal transition instantiations in an unfolding. To see this, let us return to the example shown in Figure 23 and suppose that transition a_1+ is removed from the STG together with places $p3$ and

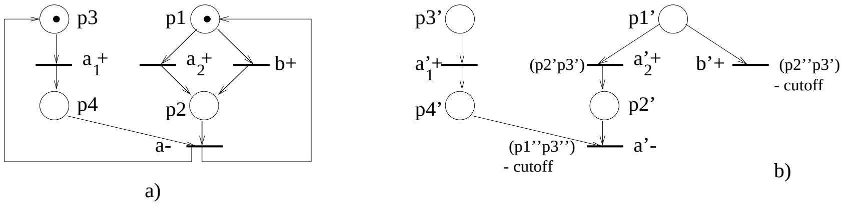


Figure 23: An STG with an inconsistent state assignment (a) and its unfolding (b)

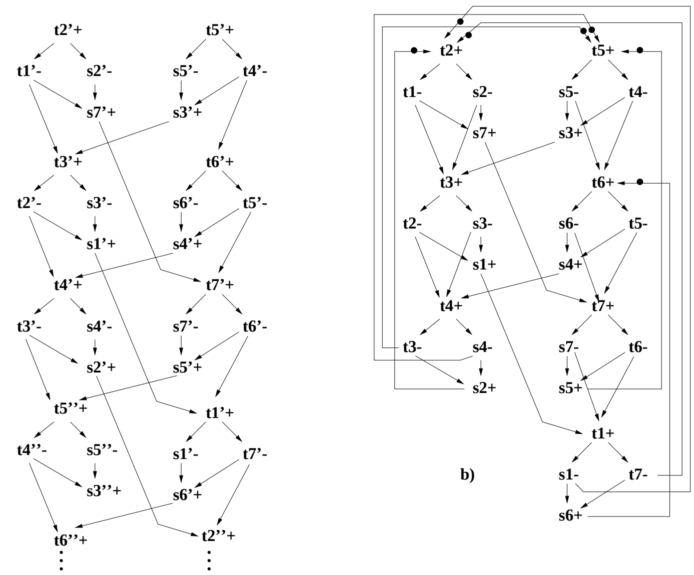


Figure 24: Inconsistent a) and consistent b) specifications of railway system

$p4$. The remaining part of the unfolding is both non-auto-concurrent and sign alternating, but the feasible trace $b+, a-, b+$ shows a violation of consistency. This fact cannot be explicitly observed in the unfolding, since there is no information about the next transition of signal b after $b'+$.

The problem can be solved by considering binary states corresponding to basic markings [27]. Each marking m' in an unfolding of an STG is mapped to one binary state s. Values of all the signals in s can be obtained by firing all the transitions from the configuration C' that corresponds to the marking m'. We will say that an STG satisfies the *proper state assignment* property if for any pair of transitions with equal basic markings, the corresponding binary states coincide.

The following statement gives necessary and sufficient conditions for an STG to be consistent.

Let D be a bounded STG *and D' be an unfolding of D. If D' is non-auto-concurrent, sign alternating and has proper state assignment, then D is consistent.*

The proper state assignment property helps to distinguish the consistency violation in the unfolding from Figure 23,b. If the initial state of STG is 00, then the firing of a_2+' or $b+$ transitions reaches the same marking, but the corresponding binary states are different: 10 and 01 respectively.

We have reduced checking STG consistency to the analysis of ordering relations in an unfolding. Indeed auto-concurrency and sign alternation are particular cases of concurrency and precedence relations, while binary states corresponding to basic markings can be calculated locally from the binary states corresponding to the preceding transitions.

Complete State Coding. This property is the most difficult in the analysis by unfoldings. CSC conflicts are essentially defined by pairs of binary states and there is little hope to check the CSC property without completely analyzing the reachability set.

To alleviate somehow this difficulty, unfolding methods can at least perform a *conservative check* of CSC [41]. This analysis is based on an approximation of the reachable state space, by testing sufficient conditions which ensure the absence of CSC conflicts. In case the conditions are satisfied, one can be sure that the STG has CSC property. Otherwise, it may either be the case that a CSC conflict really exists, or that the conditions were overly conservative (i.e. the analysis can produce false negative results).

For the exact analysis of the CSC property, it is currently better to use the BDD-based approach described in the previous section.

This Section considered the application of the unfolding technique to the verification of specific properties related to STG implementability. While unfolding-based analysis is much more powerful[16], we restricted our consideration to this special case, since its verification is simpler than the general case, but it is sufficient to decide the applicability of our synthesis technique.

7.4 Event-based vs state-based verification

Sections 6 and 7 presented two different approaches for the verification of asynchronous systems. Although both approaches are aimed at checking the same properties, they have quite different characteristics in applications.

It is not easy to provide an exact recipe where to apply any of these methods. Below we will try to give some hints, by comparing the verification results obtained by two software tools: petrify [5] based on BDD techniques and unfolding [26] based on unfolding methods.

First, let us note that for small/moderate examples both methods are nearly equally good and the analysis times are small.

[16] Any property that can be expressed by propositional logic extended with a possibility operator can be checked by unfoldings [13].

Table 2 presents the results of checking the STG implementability for the examples from Section 4.

Example	SI-impl.	SI-gate-impl.	CPU_{BDD}	CPU_{unf}	Remarks
set-dominant latch (Fig. 2,c)	+	+	0.16	0.1	< 1
consumer-producer (Fig. 5,a)	+	–	0.98	< 1	non-CSC
6-section railway (Fig. 7,b)	+	+	12.1	< 1	
7-section railway (Fig. 7,c)	–	–	27.1	< 1	Inconsistent
Modified 7-section railway (Fig. 24,b)	+	+	52.6	< 1	
3 dining philosophers (Fig. 8,b)	+	+	10.1	< 1	

Table 2: Verification results for small examples (CPU times in seconds).

To compare the efficiency of the event-based and state-based verification, let us consider the scalable examples of the railway system with n sections and of the n dining philosophers. The CPU times and the sizes of internal representations (BDD in case of state-based verification and unfolding for event-based methods) are presented in Table 3.

		# of	# of	# of	BDD		Unfolding		
Example	n	places	trans.	states	size	CPU (sec.)	# of places	# of trans.	CPU (sec.)
n sections railway	8	56	32	1.5×10^3	3091	100	106	58	< 1
	9	63	36	4.3×10^3	9185	235	128	70	< 1
	10	70	40	1.2×10^4	7073	387	150	82	< 1
	11	77	44	3.0×10^4	29623	973	172	94	< 1
	12	84	48	7.5×10^4	23042	1313	194	106	< 1
	13	91	52	1.8×10^5	55559	2794	216	118	< 1
	14	98	56	4.4×10^5	60148	4211	238	130	< 1
	15	105	60	1.0×10^6	106089	7683	260	124	< 1
	16	112	64	2.4×10^6	106308	> 24 hours	282	154	< 1
n dining philosophers	5	45	30	7.8×10^3	1032	39	462	347	1
	6	54	36	4.7×10^4	1274	77	824	626	2
	7	63	42	2.8×10^5	1512	125	1388	1052	6
	8	72	48	1.6×10^6	1752	194	2143	1627	12
	9	81	54	1.1×10^7	1992	285	3139	2386	48
	10	90	60	6.2×10^7	2232	402	4410	3354	3293
	20	180	120	3.7×10^{15}	5227	3911	–	–	> 24 hours

Table 3: Experimental results for big size examples

Table 3 clearly shows the difficulties of a direct comparison of state and event-based methods: the unfolding-based method is superior to the BDD-based technique for the verification of the railway system, while BDD traversal is more efficient for the analysis of the dining philosophers example (for cases with more than 10 philosophers).

There is some intuition behind the inefficiency of unfolding methods for the verification of the dining philosophers. The common parts of alternative branches of the PN are duplicated in the unfolding, and, roughly speaking, the unfolding grows "in width" with respect to the amount of conflicts. Because of the concurrency, different instantiations of the same PN transition can have different basic markings,[17] and hence due to concurrency the unfolding grows "in depth".

[17] In a sequential process all instantiations of the same transition always have the same basic marking.

The dining philosophers example is characterized by a complicated structure of conflict and concurrency relations. Hence the size of the unfolding quickly blows up when the number of philosophers increases. Note that this difficulty arises only in the refined version of the dining philosophers problem (when returning the left and right forks are considered as independent actions). In the classical version (when forks are returned *simultaneously*) there is no such concurrency in the behavior of a single philosopher, and the unfolding method works more efficiently than the BDD-based technique.

The above observation about the class of specifications "difficult" for the unfolding approach is far from rigorous. Things are even more vague when talking about the efficiency of BDD-based methods. The size of the BDDs obtained during a symbolic traversal can be quite different and depend on many factors. One of the most important factors is the order of signals during BDD construction: the size of BDD for the same STG with a "bad" order of signals can be exponentially larger than that with a "good" ordering [24]. However, even using the same heuristics on the ordering of signals, the size of the BDDs can differ by several orders of magnitude for STG specifications that are approximately equal in size. BDDs are more suitable for regular specifications (like our scalable examples) and can handle large examples, while for "unstructured" specifications the BDD size generally blows up faster.

The conclusion from the comparison of the unfolding and BDD approaches can be as follows: they are complementary techniques which allow one to attack from different sides the verification problem of asynchronous systems. Their efficiency depends on different and unrelated features of the specification, and if one technique fails, the other can succeed.

8 Synthesis of asynchronous circuits from STGs and PNs

Two approaches for designing asynchronous circuits using STGs, syntax-directed compilation and the logic synthesis, are shown in Figure 25.

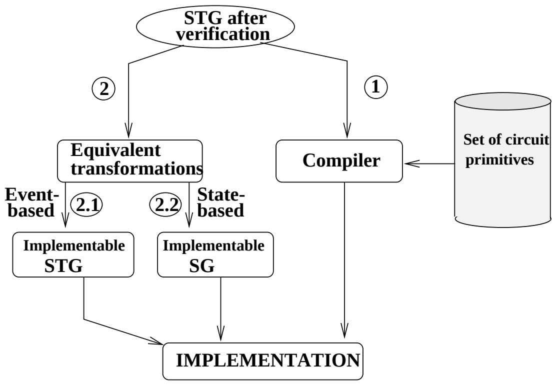


Figure 25: Design using STG specifications

Syntax-directed or structural compilation compiles an STG (or a PN) into a circuit using a syntax-directed translation to a basis of pre-designed primitives. The circuit primitives model the behavior of STG fragments (single places, single transitions or possibly larger fragments of STG). The compilation process solves two tasks: (a) matching the STG elements with corresponding primitives from a library and (b) assembling primitives into a proper implementation. This approach is fast and simple. It gives circuit implementations that are linear in the size of the original STG. The latter allows us to estimate the complexity of the implementation at the early stage of the design process. However, the implementation is often inefficient, both in terms of area and performance. This restricts the application of the structural synthesis methods.

The synthesis-based approach in in most respects opposite to the structural methods: it is complex and computationally expensive, but it gives much more efficient implementations. In this approach the initial specification is gradually refined until the behavior of each signal can be implemented by a library gate. The refinement is done by means of equivalent transformations (e.g., by inserting new signals) and can be performed either directly at the STG level (route 2.1 in Figure 25) or using the corresponding SG (route 2.2 in Figure 25).

8.1 Structural synthesis

There are two basic elements in a PN: places and transitions. Simulating the PN behavior by a circuit can be done using either places or transitions as a modeling basis.

Modeling by places. For simplicity let us consider safe and persistent PNs. The goal of the synthesis procedure is to achieve a correspondence between the marking of every place p and the value of the output signals corresponding to p. The binary value corresponding to the marking of p ("0" if there is no token, "1" if there is a token) can be stored in a latch. When the marking of p is supposed to change according to the token flow, the corresponding latch changes the stored value.

An example of circuit primitives for the place-based implementation of PN is shown in Figure 26,a [58]. The marking of any place p is encoded by the values at the outputs of the RS-latch, (p, p'), as follows: (0,1) – no token ("p is not marked"), and (1,0) – there is a token ("p is marked"). $a1$ is the "activity gate" which triggers the transmission of a token from place p to the following places.

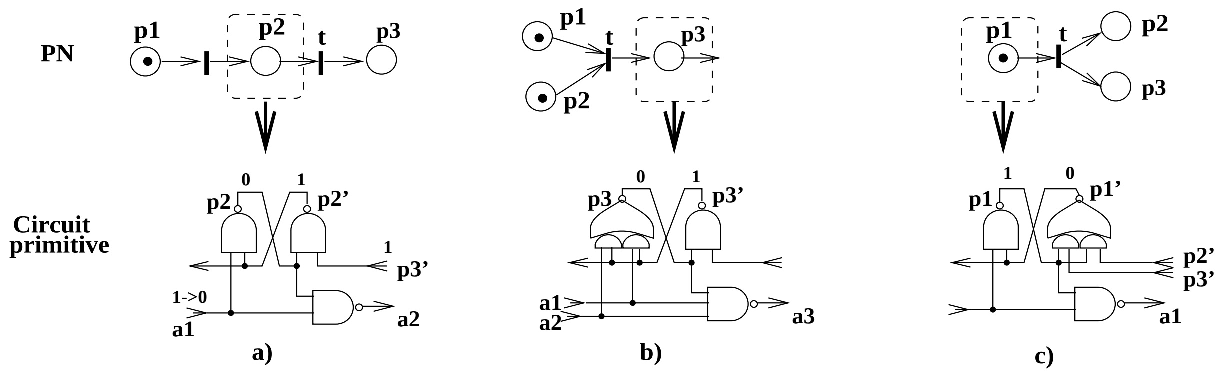


Figure 26: Place-based implementation of PN

The token flow in the PN fragment in Figure 26,a is simulated by the circuit in the following way. Initially place $p1$ is marked (latch $(p1, p1')$ is in state (1,0)) while $p2$ is not (latch $(p2, p2')$ is in state (0,1)). Passing the token from $p1$ to $p2$ starts by the switching $a1-$ of the activity gate $a1$. This forces the latch $(p2, p2')$ to change the state from (0,1) (unmarked) to (1,0) (marked) going through the transient state (1,1) via events $p2+$,$p2'-$. The latter triggers the latch $(p1, p1')$ to move into the unmarked state (events $p1'+$,$p1-$) which is acknowledged at the cell modeling $p2$ by switching the activity input $a1$ (event $a1+$). The token has shifted from $p1$ to $p2$.

From this consideration, it follows that the atomic firing operation of transition t is modeled in the circuit in two steps: first the token appears in $p2$ and then it is erased in $p1$. However, this does not lead to any confusion in the modeling process, because in the transient state (when the token resides in both $p1$ and $p2$) the token flow is stopped by preventing the activity gate $a2$ from switching (gate $a2$ can be set only when latch $(p1, p1')$ goes to the unmarked state).

The basic primitive can be easily modified for the modeling of more complicated token flows. For $p3$, that is an output place of the *join* transition t (see Figure 26,b), a token can appear only when both $p1$ and $p2$ are marked. This is modeled by AND-ing falling events, $a1-$ and $a2-$, from the activity gates of the cells for $p1$ and $p2$. For $p1$, that is an input place of the *fork* transition t (see Figure 26,c), the token is erased only when it appears in both $p2$ and $p3$. This is modeled by AND-ing the falling transitions of signals $p2'$ and $p3'$ for resetting latch $(p1, p1')$. Note that the restriction of safeness is not crucial for the suggested approach, since unsafe nets can be modeled in a similar way by adding additional buffers and pielining the basic primitives. Non-persistency can be modeled by adding additional input variables from the environment.

By thorough analysis of a PN structure it might be possible to simplify the modeling circuit [57]. However, there are few hopes to get an area efficient circuit using this approach because of the low level of granularity in modeling PNs: in general it is too expensive to model each place with a separate latch. Still, this approach might be useful for fast prototyping of asynchronous controllers when the synthesis time (not the area or delay cost) is a crucial issue.

Modeling by transitions. The alternative way of simulating the PN behavior by a circuit consists of modeling by transitions. In the simplest form, the firing of transition t is modeled by the switching of the output T of the corresponding circuit primitive. The direction of switching (rising or falling) is irrelevant. Hence, if initially the output T is at 0 every odd firing of transition t corresponds to $T+$ while every even firing corresponds to $T-$. Therefore, this kind of modeling is called *two-phase*.

The first complete set of circuit primitives for two-phase modeling of PN behavior was suggested by Patil and Dennis [44]. The constructions presented there are simple and elegant, but they are suitable only for those safe, persistent nets in which at most one input place can be shared by several transitions. These restrictions are partly removed (safeness is still necessary) in the modeling scheme of Figure 27 [22].

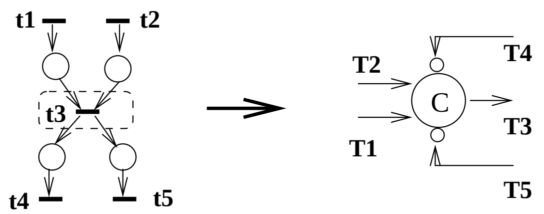


Figure 27: Transition-based implementation of PN

Firing of transition $t3$ corresponds to the switching of a four-input C-element $T3$. If initially signals $T1, T2, T3, T4, T5$ are all at 0, then output $T3$ will rise only after rising of signals $T1$ and $T2$ which model firings of transitions $t1$ and $t2$. The next occurrence of transition $t3$ is modeled by signal $T3$ falling. The latter happens only after the falling of $T1$ and $T2$ (transition $t1$ and $t2$ must fire for the second time) and rising of $T4$ and $T5$ (the successors of $t3$ must finish their first firings). Hence the negative feedbacks from the circuits modeling successor transitions prevent the interference of consecutive firing flows for transition $t3$.

The transition-based implementation is typically more area efficient than the place-based one, but it is still less efficient than the logic syntehsis techniques described in the next Section.

8.2 Synthesis-based methods

Section 5.1 gave the formal conditions which must be satisfied by an STG in order to be implementable. In particular, it was shown that if a deterministic, persistent and consistent STG with n non-input signals satisfies the CSC property, then there exists a speed-independent circuit with n gates, whose behavior is strongly equivalent to the original STG. However, this statement does not restrict in any way the types of gates used in such circuit. It assumes that the implementation basis is not limited and that the gates can implement *any arbitrary* Boolean function (so-called complex gates). Even though this assumption looks rather unrealistic, methods for complex gate implementation are of interest due to the following reasons:

1. A complex gate implementation can satisfy the requirements of real library (we will see that this is the case with several examples that we have considered so far).
2. A complex gate implementation is a good starting point for technology mapping methods which decompose complex gates into library blocks [31, 51, 25].
3. With the development of ASICs and FPGAs the implementation basis can realize quite complex Boolean functions (e.g., the cell of a Xilinx FPGA can implement any Boolean function of up to 5 variables [19]).

The task of complex gate implementation of an STG with the CSC property is to obtain logic equations for all its non-input signals. It has a straightforward solution if one considers the corresponding SG.[18]

If the Complete State Coding requirement is met, then for any non-input signal of the STG, the SG defines an *incompletely specified logic function* as three sets of Boolean vectors: the *on-set*, the set of states where the function evaluates to 1; the *off-set*, the set of states where it evaluates to 0; and the *dc-set* (don't care set), the set of states where the function is not specified.

The dc-set includes all the states which are not reachable from the initial state of the SG. We can define which states belong to the on-set and off-set by considering the so-called implied value of a signal, i.e. the next value that the signal is going to take.

If signal a is *stable* in the SG state s, then s belongs to the off-set of a if a=0 in s and to the on-set of a if a=1 in s.

If signal a is *enabled* in state s, then its value must be the opposite to the current one, and s belongs to the off-set of a when the current value of a is 1 and to the on-set when it is 0.

Using the "*" notation for enabled signals, the above rules can be shortly formulated as follows: if $a = 1 \vee a = 0^*$ in state s, then s is included in the on-set of a, while if $a = 0 \vee a = 1^*$, then s is included in its off-set.

Example. Let us illustrate the synthesis procedure by deriving the logic equation for the set-dominant latch from Section 4. Its behavior specification in the form of SG in Figure 28,a is reproduced from Figure 2,d. The output signal Q is equal to 0 and is stable in the initial state 0*1*0 and hence this state belongs to the off-set of Q. Q keeps the 0 value but is enabled in 110*, and 110* is included into the on-set of Q. Following these rules we can fill the Karnaugh map for signal Q (shown in Figure 28,b) and derive the logic equation $Q = S \vee \overline{R}Q$ with conventional minimization techniques. This is the well-known equation of an SR-latch.

[18] Synthesis can be performed directly from the STG as well. These methods are based on approximation techniques similar to the one discussed in Section 7.3. However, they are more complicated and we will omit them due to the lack of space. One can look for details in [43].

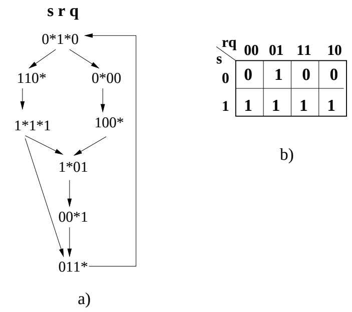


Figure 28: Derivation of logic equations for set-dominant latch

The circuits for the semaphore control of the railway system with 6 and 7 sections, automatically derived using the above method (see Figures 7,b, 24,b and Table 2), are shown in Figure 29,a,b. The overall circuits for the railway control are shown in Figure 29,c and d respectively. One can see that for both circuits the parameters of the complex gates are quite reasonable and these implementations can be realized in most real libraries.

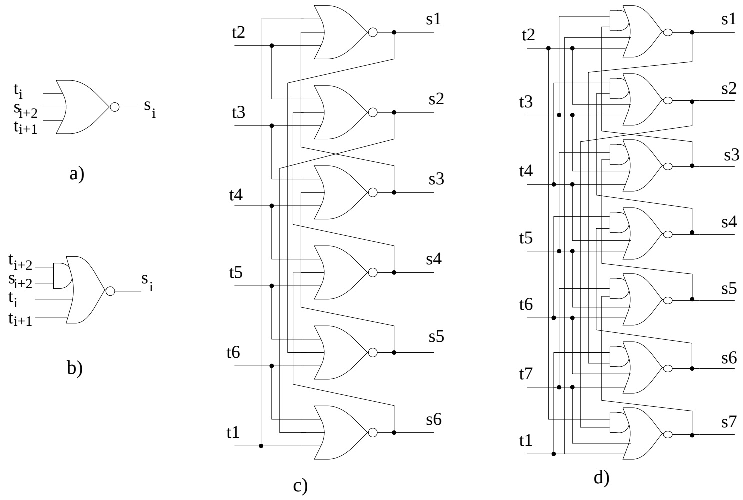


Figure 29: Implementations of the railway system control

If an STG is not SI-implementable, then there is no speed-independent circuit that is trace equivalent to the STG. Such specifications are rejected by the synthesis procedure and must be changed by hand to satisfy the implementability properties.

We next describe a method to satisfy the CSC requirement by inserting additional state signals.

8.3 Complete State Encoding

When an STG contains CSC conflicts, they must be disambiguated by additional signals. This can be done by inserting one signal at a time until all conflicts are resolved[19]. Such an iterative procedure can be formulated as follows:

1. Partition SG states into two subsets S^0 and S^1. The inserted signal will have value 0 in all states of set S^0 and value 1 in all states of set S^1.

[19]Vanbekbergen *et al.* first formulated a similar procedure in [56] but attempted to solve all conflicts at once. This is suboptimal and can fail to find a feasible solution in some cases.

2. Insert a new signal to switch "between" the states from S^0 and S^1. For this, find subsets of states $S^+ \subset S^0$ and $S^- \subset S^1$ such that the inserted signal is enabled to fire up and down in all states of S^+ and S^-, respectively.

3. Check the new SG for CSC conflicts, iterate if CSC conflicts are found.

Before introducing the theory behind the above procedure, let us consider informally how it works for the synthesis of the consumer-producer system from Figure 5.

Example. There is one CSC conflict for states 0*0*01 and 0001* in the SG, and we can separate the conflicting states by the partition shown in Figure 30,b.

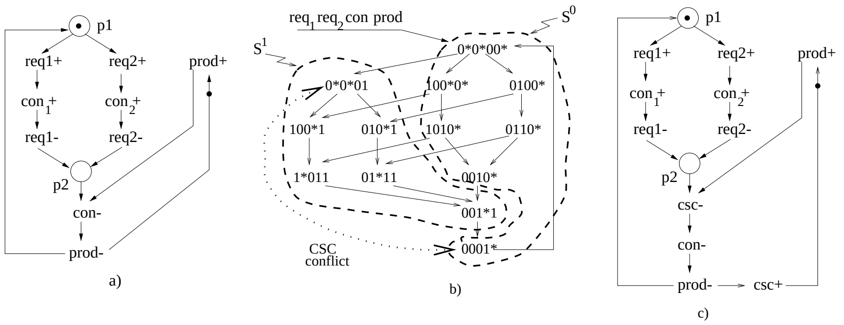


Figure 30: Reduction to the CSC form

State 0011 is the only state of S^1 from which one can enter the set S^0, hence it is the only state in which transition $csc-$ of the new signal csc is enabled (see the insertion of $csc-$ in the STG in Figure 30,c). On the other hand, there are many border states by which we can enter S^1 from S^0, and signal $csc+$ should be enabled in all of them. This means that transition $csc+$ must be inserted concurrently with the transitions between border states (see Figure 30,c). The STG obtained after the insertion of signal csc satisfies the CSC property and its implementation on complex gates is shown in Figure 31,a. One can see that the logic functions of the implementation are more complicated than in the case of railway system controller, and their complexity may exceed the requirements of the gate library. The complex gate implementation can be decomposed into simpler gates and C-latches by the technology mapper [5](see Figure 31,b).

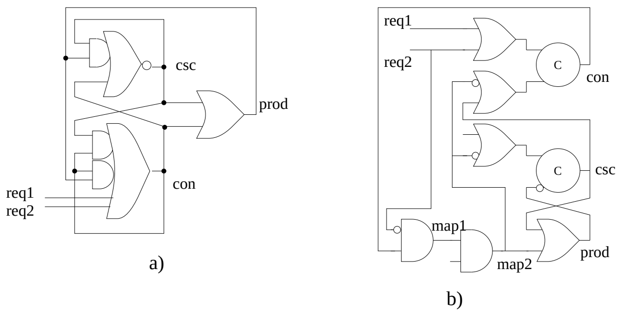


Figure 31: Implementation of control for consumer-producer system

Two questions are important for the above CSC elimination procedure:

- How to find the bi-partition of SG states into sets S^0 and S^1?

- How to find the "proper" set of states for enabling the inserted signal such that speed-independence (both of old and newly inserted signals) is preserved?

We will answer these questions in the next section by using the theory of state regions.

8.4 Speed-independence preserving event insertion

Regions, Excitation Regions and Switching Regions. Let r be a subset of the states of an SG $r \subseteq S$. If $s \notin r$ and $s' \in r$, then we say that transition $s \xrightarrow{a} s'$ *enters* r. If $s \in r$ and $s' \notin r$, then transition $s \xrightarrow{a} s'$ *exits* r. Otherwise, transition $s \xrightarrow{a} s'$ *does not cross* r.

A *region* [40] is a subset of states with which *all* transitions labeled with the same event e have exactly the same "entry/exit" relation. This relation will become the predecessor/successor relation in the Petri net.

Let us consider the SG shown in Figure 32. The set of states $r1$ is a region, since all transitions labeled with $a+$ enter $r1$, all transitions labeled with $b-$ exit $r1$, and transitions labeled with $b+$ and $a-$ do not cross r. On the other hand, the set of states $\{1{*}1{*},\ 1{*}0\}$ (shown by the dotted line in Figure 32) is not a region since transition $0{*}1 \xrightarrow{a+} 1{*}1{*}$ enters this set, while another transition labeled with $a+$, $0{*}0{*} \xrightarrow{a+} 10{*}$, does not.

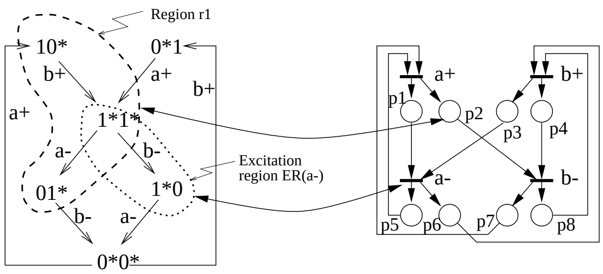


Figure 32: Regions and excitation regions in SG

A region r is a *pre-region* of event e if there is a transition labeled with e which exits r. A region r is a *post-region* of event e if there is a transition labeled with e which enters r.

Figure 32 shows the correspondence between regions and Petri net places. E.g., place $p2$ corresponds to region $r1$. Place $p2$ is an input place for the transition labeled with $b-$ because region $r1$ is a pre-region of $b-$.

While regions in an SG are related to places in the corresponding PN, excitation regions are related to transitions of the PN.

A set of states is called an *excitation region* for event a (denoted by $ER(a)$) if it is a *maximal* set of states such that for every state $s \in ER(a)$ there is a transition $s \xrightarrow{a}$.

It can be shown [6] that in the SG corresponding to a safe STG, the excitation region of a persistent transition can be obtained as the intersection of all pre-regions for this transition.

In the SG from Figure 32 the excitation region for $a-$, $ER(a-)$, is shown by the dotted line. It corresponds to the transition $a-$ in the PN.

Similarly to the ER, we define the switching region for event a as the set of states reached *immediately after* the occurrence of a (denoted by $SR(a)$).

Insertion scheme. Informally, the event insertion operation selects a subset of states, splits each one of the selected states into two states and creates, on the basis of these new states, an excitation and a switching region for the new event. Figure 33 illustrates the insertion scheme which is typically used in applications to asynchronous design.

A state signal insertion must preserve the SI-implementability properties of the original specification. Formally, we say that a set of states $ER(x)$ selected for the insertion of a new signal in an

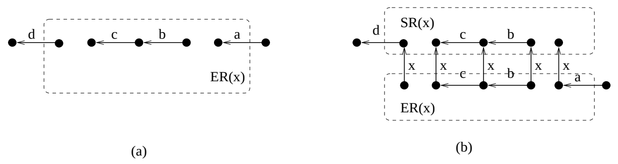


Figure 33: Event insertion technique

SG G is a *speed-independence preserving set (SIP-set)* iff the SG G' obtained from G by inserting event x is SI-implementable. A set of states r is SIP iff it intersects any state diamond according to one of the cases shown in Figure 34. A state diamond is a quadruple of states $s1, s2, s3, s4$ such that two concurrent events a and b can fire between those states as shown in Figure 34,a [5].

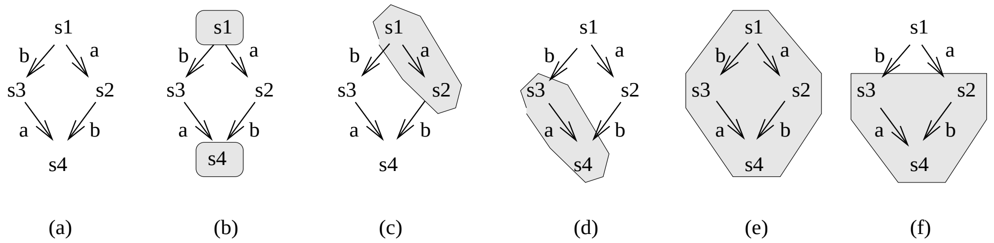


Figure 34: SIP-sets for a state diamond

The following properties of insertion sets link together *regions* and *SIP-sets*. It can be shown [5] that in an SI-implementable SG a set of states r is SIP if:

1. r is a *region* or
2. r is an *excitation region* of a persistent event a or
3. r is an *intersections of pre-regions* of the same event, when r is connected and all its exit events are persistent.

This suggests that good candidates for insertion sets should be sought on the basis of regions and their intersections. Since any disjoint union of regions is also a region, this gives an important corollary that sets of states selected for insertion of new events can be built more efficiently from "bricks" (regions) rather than "sand" (states).

Partitions and borders. We now return to the first problem of bi-partitioning the set of states. To distinguish the CSC conflicts in an SG, a bi-partition of its states into sets S^0 and S^1 should be constructed. Even though the conflict relation is defined on pairs of states, the requirements of SIP-sets do not allow us to manipulate single states in the construction of a partition. The problem of finding an optimal SIP partition maximizing the number of distinguished conflicts is intractable, and hence requires to develop appropriate heuristics.

Such an approach can be based on considering larger sets of states when constructing a partition. Regions and intersections of pre-/post-regions of the same event are good candidates for "bricks" when looking for SIP sets, rather than individual states ("sand"). To construct a partition, adjacent bricks are combined together to form bigger blocks. The union of bricks is guided by a cost function that takes into account the number of distinguished conflicts. A greedy block merging approach can be used in practice.

Example. The SG in Figure 32 has two pairs of CSC conflicts {10*, 1*0} and {0*1, 01*}. A partition distinguishing the conflicts is constructed from the intersections of post-regions. The post-regions of transition $a+$ are regions $r1$ and $r2$ (shown in Figure 35 by dashed lines), and those transition $b+$ are $r3$ and $r4$ (shown by dotted lines). Their intersections are shown by dashed areas in Figure 35 and the union of intersections gives set S^0. The states not included into S^0 form S^1, i.e., $S^1 = S - S^0$.

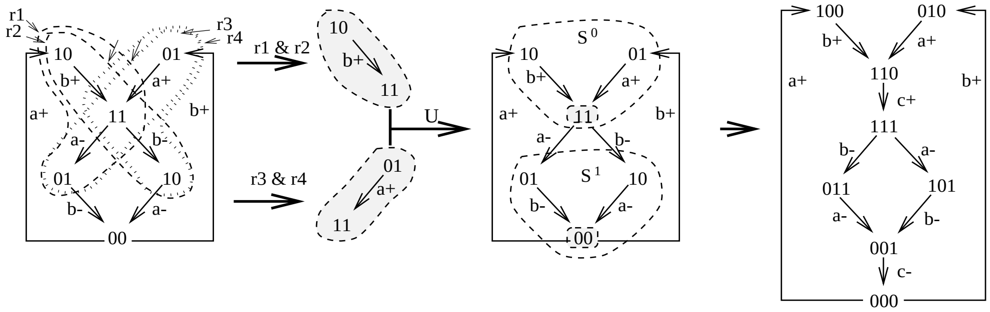


Figure 35: Elimination of conflicts by the region-based approach

Given a bi-partition of the set of SG states into S^0 and S^1, the excitation regions for the new signal should be found. The new signal should change its value when crossing the borders between S^0 and S^1. Hence exit (or input) borders of S^0 and S^1 are the first candidates for the excitation regions. Intuitively, the *exit border*, of a partition block r, denoted by $EB(r)$, is a subset of states of r from which there are some transitions exiting r. We call $EB(r)$ *well-formed* if there are no transitions leading from states in $EB(r)$ back to states in $r - EB(r)$. If the exit border is not well-formed, then after insertion the consistency of state assignment will be violated. Hence well-formedness is a necessary condition for preserving the SI-implementability properties.

Given a partition $\{S^0, S^1\}$ it is always possible to find the minimal well-formed exit borders (denoted by $\mathsf{MWFEB}(S^0)$ and $\mathsf{MWFEB}(S^1)$). For example, $\mathsf{MWFEB}(S^0)$ can be calculated as the least fix point of the following recursion:

1. $\mathsf{MWFEB}(S^0) = EB(S^0)$
2. $[s \in \mathsf{MWFEB}(S^0) \ \wedge \ s' \in S^0 \ \wedge \ s \to s'] \ \Rightarrow \ s' \in \mathsf{MWFEB}(S^0)$

Example. For the partition $\{S^0, S^1\}$ of the SG in Figure 32, state 1*1* is the only one from which there is a transition exiting S^0, hence $EB(S^0)$ ={1*1*}. Similarly, $EB(S^1)$ ={0*0*}. Both exit borders (shown by the dashed areas) are well-formed and satisfy the SIP-set requirements. The result of the insertion of the additional signal c using these exit borders is shown at the right in Figure 35. The resulting SG has no CSC conflicts and can be implemented by a speed-independent circuit. This circuit corresponds to the C-latch with two inputs inverters that we have already discussed in Section 4 (see Figure 10).

In the example of Figure 32, the obtained bipartition distinguishes all CSC conflicts in the original SG. In general, several iterations inserting signals can be required. On every iteration the specification is refined by excluding some conflicts. The convergence of the procedure was shown in [5] for the class of SGs corresponding to safe STGs.

9 Conclusion

This paper has discussed various techniques for specification, analysis and synthesis of asynchronous circuits that have been developed in recent years based on an interpreted PN specification called STG. We have shown, by means of examples, how the function of an asynchronous *control* circuit can be specified using an STG, and how an existing asynchronous netlist can be modeled (for analysis and re-synthesis) as an STG. A data path is usually designed independently and is controlled by the control circuit via some form of request/acknowledge or using timing assumptions (bundled data) to simplify the control logic.

We have compared BDD-based versus unfolding-based analysis techniques for STGs. Finally, we have summarized state encoding and logic synthesis techniques for asynchronous circuits using STGs.

We hope that this review has been useful, first of all as a contribution towards the establishment of a practical asynchronous circuit design flow, second as a paradigm for the establishment of other interpretation of Petri nets for *synthesis* of other classes of asynchronous processes. A designer interested in applying these synthesis techniques should use the examples presented in this paper as a guideline to specify asynchronous controllers by means of STGs, and use the tools based on the theory described in the previous sections in order to implement the control logic. An example of such tools is `petrify`, which is available from *http://www.ac.upc.es/vlsi/petrify/petrify.html*. The data path, on the other hand, could be designed by using micropipelining or semi-synchronous techniques, in which the clock signals are locally generated by the controllers [52, 15]. Standard physical design tools can then be used, with the additional constraints arising from the need to satisfy the isochronic fork assumptions, to place and route the circuit. Information about such isochronic forks can be obtained directly by using PN-based verification techniques.

We strongly believe that the scope of application of PNs should not be limited to analysis (either by simulation or by means of formal techniques) but be extended to *synthesis* as well. This is possible only by mapping the interpretation of the net onto a domain for which synthesis techniques can be developed, such as the Boolean domain in this case, and by interpreting PN properties, such as persistence, as desirable (or dangerous) implementation properties, such as hazard-freeness.

In the future, we will need to enhance the techniques summarized in this paper with some notion of time, in order to compete with synchronous circuits that take full advantage of the fact that physical delays are always bounded. Although some work in this area has been done ([38]), a more aggressive or/and more efficient optimization is still needed. Moreover, a good design flow needs an abstraction mechanism, and unfortunately general PNs are particularly lacking in this respect (mostly because logic circuit abstraction is based on "boxes" with inputs and outputs, while an STG only models sequencing of events). Again, some work has been done ([59]), but it needs more development.

Acknowledgments

We would like to thank, among many others, Javier Esparza, Enric Pastor, Marco Pena, Oriol Roig, Alex Semenov, Sergei Ten, Peter Vanbekbergen, Alexandre Yakovlev, for many useful discussions of the topics in this paper. We thank Steve Furber and anonymous reviewers for useful comments on the draft of this paper.

References

[1] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, 1994.

[2] Randal Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[3] S. Burns. General conditions for the decomposition of state holding elements. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.

[4] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

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

[6] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. In *Proceedings of the International Conference on Computer-Aided Design*, November 1995.

[7] J. Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev. Designing asynchronous circuits from behavioural specifications with internal conflicts. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah*, November 1994.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[9] J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, Cambridge, Great Britain, 1995.

[10] V. Diekert. Combinatorics on traces. In *Lecture Notes in Computer Science*, volume 454. Springer-Verlag, 1987.

[11] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[12] A. Ehrenfeucht and G. Rozenberg. Partial 2-structures; part I: Basic notions and the representation problem, and part II: State spaces of concurrent systems. *Acta Informatica*, 27:315–368, 1990.

[13] J. Esparza. Model checking using net unfoldings. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th Int. Joint Conference CAAP/FASE*, volume 668 of *Lecture Notes in Computer Science*, pages 613–628. Springer-Verlag, 1993.

[14] J. Esparza. Cutoff criteria for unsafe Petri nets. Personal communication, 1997.

[15] S. Furber. Computing without clocks: Micropipelining the ARM processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[16] P. Godefroid. Using partial orders to improve automatic verification methods. In E.M Clarke and R.P. Kurshan, editors, *Proc. International Workshop on Computer Aided Verification*, 1990. DIMACS Series in Discrete Mathematica and Theoretical Computer Science, 1991, pages 321-340.

[17] M. Hack. Analysis of production schemata by Petri Nets. Technical Report TR 94, Project MAC, MIT, 1972.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[19] Xilinx Inc. *The XC4000 Data Book.* 1993.

[20] S. Römer J. Esparza and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Lecture Notes in Computer Science, Vol. 1055.* Springer-Verlag, 1996.

[21] J. R. Jump and P.S. Thiagarajan. On interconnection of asynchronous control structures. *Journal of the ACM*, 22(4):102–112, 1975.

[22] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* John Wiley and Sons, London, 1993.

[23] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. On self-timed behavior verification. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1992.

[24] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *Proc. of European Design and Test Conference*, pages 325 – 332, Paris(France), March 1995.

[25] A. Kondratyev, J. Cortadella, L. Lavagno M. Kishinevsky, and A. Yakovlev. Technology mapping for speed-independent circuits: decomposition and resynthesis. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eindhoven*, April 1997.

[26] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. A structural approach for the analysis of Petri nets by reduced unfoldings. In *International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Sciences*, Osaka, Japan, June 1996.

[27] Alex Kondratyev and Alexander Taubin. Verification of speed-independent circuits by STG unfoldings. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75, November 1994.

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

[29] C. Y. Lee. Representation of switching functions by binary decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[30] K.-J. Lin and C.-S. Lin. A realization algorithm of asynchronous circuits from STG. In *Proceedings of the European Design Automation Conference (EDAC)*, pages 322–326, 1992.

[31] F. Mailhot and G. De Micheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Transactions on Computer-Aided Design*, 12(5):599–620, May 1993.

[32] K. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[33] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Asynchronous design for programmable digital signal processors. *IEEE Transactions on Signal Processing*, 39(4):939–952, April 1991.

[34] R. E. Miller. *Switching theory*, volume 2, chapter 10, pages 192–244. Wiley and Sons, 1965.

[35] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In *Chapel Hill Conference on VLSI*, pages 67–86, May 1985.

[36] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

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

[38] C. Myers and T. H-Y. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the International Conference on Computer Design*, October 1992.

[39] M. Nielsen, G. Plotkin, and Winskel G. Events structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1980.

[40] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.

[41] E. Pastor and J. Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proceedings of the International Conference on Computer-Aided Design*, November 1993.

[42] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, Zaragoza, Spain, June 1994.

[43] Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural methods for the synthesis of speed-independent circuits. In *Proc. of European Design and Test Conference*, pages 340 – 347, Paris(France), March 1996.

[44] S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.

[45] J. L. Peterson. *Petri Nets*, volume 9. ACM Computing Surveys, No. 3, September 1977.

[46] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).

[47] L. Pomello, G. Rozenberg, and C. Simone. A survey of equivalence notions for net based systems. *Lecture Notes in Computer Science*, 609:410–472, 1993.

[48] W. Reisig. *Petri Nets: an Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[49] Oriol Roig, Jordi Cortadella, and Enric Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Asynchronous Design Methodologies*, pages 129–137, May 1995.

[50] L. Rosenblum. The signal graph language for the modeling of exchange protocols and aperiodic circuits. In *Proc. of Conference on Simulation of digital control and computer systems*, Sverdlovsk, USSR, February 1981. In Russian.

[51] P. Siegel and G. De Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 558–565, November 1994.

[52] I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.

[53] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[54] A. Valmari. State of the art report: Stubborn sets. *Petri Nets Newsletter*, (46):6–14, 1994.

[55] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *IEEE Transactions on Computer-Aided Design*, January 1993.

[56] Peter Vanbekbergen, Bill Lin, Gert Goossens, and Hugo de Man. A generalized state assignment theory for transformations on signal transition graphs. *Journal of VLSI Signal Processing*, 7(1/2):101–115, February 1994.

[57] V. Varshavsky and V. Marakhovsky. Asynchronous control device design by net model behaviour simulation. In *International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Sciences*, Osaka, Japan, June 1996.

[58] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes.* Kluwer Academic Publisher, 1990. (Russian edition: 1986).

[59] A. V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proceedings of the International Conference on Computer Design*, pages 396–400, October 1992.

[60] A. V. Yakovlev, A. M. Koelmans, A. Semenov, and D. J. Kinnement. Modelling, analysis and synthesis of asynchronous control circuits using Petri nets. *Integration, the VLSI journal*, 21(3):143–170, December 1996.

[61] Michael Yoeli. Specification and verification of asynchronous circuits using marked graphs. In K. Voss, H. J. Genrich, and G. Rozenberg, editors, *Concurrency and Nets, Advances in Petri Nets*, pages 605–622, 1987.