Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers

Jordi Cortadella, Univ. Politécnica de Catalunya, Barcelona, Spain
Michael Kishinevsky, Alex Kondratyev, The University of Aizu, Japan
Luciano Lavagno, Politecnico di Torino, Italy
Alex Yakovlev, University of Newcastle upon Tyne, United Kingdom

Abstract

Petrify is a tool for (1) manipulating concurrent specifications and (2) synthesis and optimization of asynchronous control circuits. Given a Petri Net (PN), a Signal Transition Graph (STG), or a Transition System (TS) [1] it (1) generates another PN or STG which is simpler than the original description and (2) produces an optimized net-list of an asynchronous controller in the target gate library while preserving the specified input-output behavior. An ability of back-annotating to the specification level helps the designer to control the design process.

For transforming a specification petrify performs a token flow analysis of the initial PN and produces a transition system (TS). In the initial TS, all transitions with the same label are considered as one event. The TS is then transformed and transitions relabeled to fulfill the conditions required to obtain a safe irredundant PN.

For synthesis of an asynchronous circuit petrify performs state assignment by solving the Complete State Coding problem. State assignment is coupled with logic minimization and speed-independent technology mapping to a target library. The final net-list is guaranteed to be speed-independent, i.e., hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification. The tool has been used for synthesis of PNs and PNs composition, synthesis and re-synthesis of asynchronous controllers and can be also applied in areas related with the analysis of concurrent programs. This paper provides an overview of petrify and the theory behind its main functions.

1 Introduction

Petri nets [31, 28] are a widespread formalism to model concurrent systems. By labeling transitions with symbols from a given alphabet, transitions can be interpreted as the occurrence of events or the execution of tasks in a system. Labeled Petri Nets have been used in numerous applications: design and specifications of asynchronous circuits [34, 7, 23, 20], resource allocation problem in operating systems and distributed computation [35], analysis of concurrent programs [32], performance analysis and timing verification [19, 33], high-level design [16]. Petri Nets are popular due to their inherent ability to express both concurrent and non-deterministic behavior.

State-based models are common languages for formal specification and verification of complex systems (FSMs [15, 22], Burst mode automata [30]). Even the formal operational semantics for most of the event-based models (CSP [18], CCS [24, 25]) is given by means of states. The drawback

[1] Transition system is a directed graph with vertices labeled as states and arcs labeled with events. Transition system can be viewed as an abstract state graph.

of state-based models is that they represent causality, concurrency and conflict relations between events in terms of state sequences or state configurations (e.g., state diamonds). This is an undesirable characteristic for the designer, who always wants succinct representations of a system that explicitly represent its properties. Therefore, it is very important to identify, starting from a flat state-based representation, the set of *causality relations, concurrent events* and *conflict conditions* implicit in the representation itself, because they carry useful information for the designer or/and design algorithms.

Tool `petrify` implements a method which, given a finite state model, called Transition System (TS) in the sequel, synthesizes a safe Petri Net with a reachability graph that is *bisimilar* to the original TS. In particular, the reachability graph can be either isomorphic to the original TS or isomorphic to a minimized version of the original TS. The synthesized PN is always place-irredundant, i.e., it is not possible to remove any place from the net without changing its behavior. The synthesis technique is based on constructing *regions*. A region in a TS is a set of states corresponding to a place in a PN. Transitions in and out of this set of states "mimic" the PN firing behavior (which un-marks predecessor places and marks successor places of a transition).

The notion of regions was introduced in [17] (and developed in [29, 1, 3, 14, 26]) as a basic intermediate object between state-based and event-based specifications. This papers have been limited with the so-called class of elementary TSs which allow for PN representation with uniquely labeled transitions (each event has only one occurrence in the PN). We have shown [11] how theory of regions can be efficiently used for synthesizing place-irredundant and place-minimal PNs for elementary and non-elementary TSs.

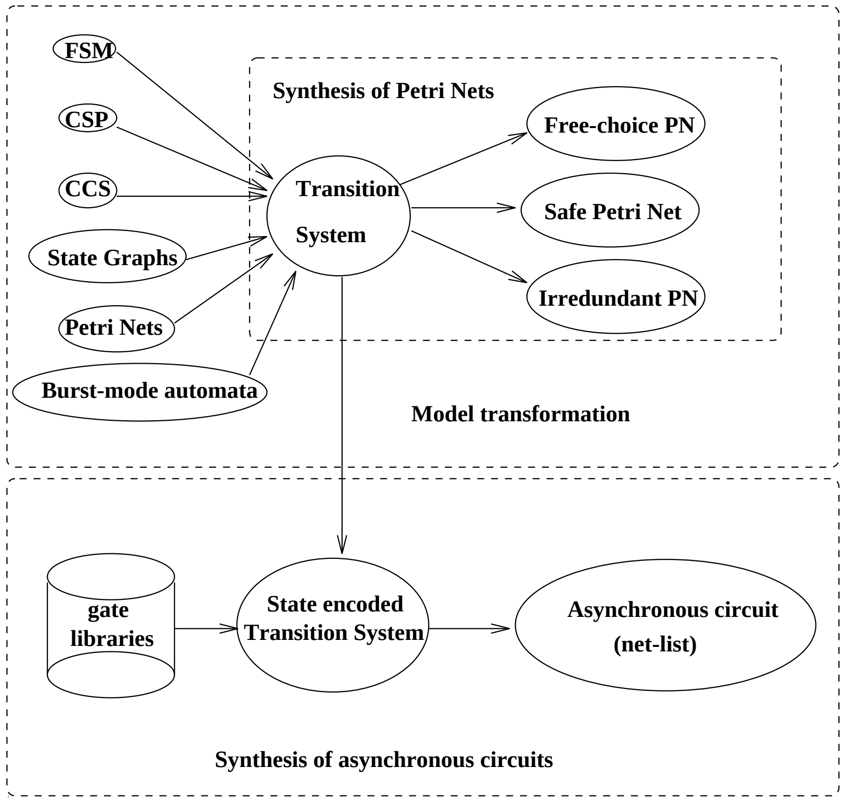


Figure 1: `Petrify`'s framework for manipulating specifications and for designing of asynchronous circuits

The method for synthesis of PNs provides a technique for transforming specifications. Given a model which can be mapped into a TS, we can derive a PN which is bisimilar to the initial model of the process. In such a way we can create a tool which automatically translates CSP, CCS, FSM, Burst-mode machines and other models into labeled Petri Nets. Also, we can use this tool for the transformation of Petri Nets aimed at optimality under some criterion (place count, transition count, number of places, PN graph complexity, etc.) or for deriving a net belonging to a given class (pure, free choice, unique choice, etc.). Such an interactive tool allows a designer to play with a PN-like specification, performing equivalent transformations of PNs, and/or transformations of other specifications into PNs under different design constraints and optimization criteria. Fig. 1 shows our framework for synthesizing PNs and transforming specifications.

In [8, 10, 9] we show that regions are tightly connected with the set of properties that must be preserved across the state encoding and technology mapping process for asynchronous circuits. Hence, regions and their intersections can be efficiently used for state signal insertion. Therefore, sets of states which correspond to places (and transitions) of PNs are useful for efficient synthesis techniques of digital circuits. For synthesis of asynchronous circuits `petrify` performs state assignment by solving the Complete State Coding problem [7, 23]. State assignment is coupled with logic minimization and speed-independent technology mapping to a target library (Fig. 1). The final net-list is guaranteed to be speed-independent, i.e., hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification.

This paper is further organized as follows. Section 2 describes what `petrify` can do in more details. Section 3 describes how `petrify` manipulates concurrent specifications. Section 4 shows to synthesize asynchronous control circuits with `petrify`. Section 5 concludes the paper and shows directions for the future development of the tool.

2 What is `Petrify`

2.1 Manipulating PNs and TSs

`Petrify` has two basic functions that allow manipulating concurrent specifications:

- *Synthesis of safe Petri Nets or Signal Transition Graphs from a given Transition System.*

  STGs are PNs with transitions interpreted as changes of the circuit signals. They are widely used in design of asynchronous circuits. TSs are abstract state graphs with labeled arcs. State Graphs are binary encoded TSs. An example of the transformation performed by `Petrify` is shown in Fig. 2.

- *Re-synthesis of Petri Nets and Signal Transition Graphs.*

  Behavior-preserving transformation of PNs can be aimed at optimality under some criterion (place count, transition count, number of places, PN graph complexity, etc.) or at deriving a net belonging to a given class (safe, Free-Choice, Unique-Choice, etc.).

  Given a bounded PN (possibly with weighted arcs and inhibitor arcs) `petrify` will generate an equivalent safe place-irredundant PN. For example, given a PN in Fig. 3(b), which corresponds to a TS from Fig. 3(a) `petrify` will produce as an output a place-irredundant (and place-minimal) safe PN shown in Fig. 3(c).

2.2 Synthesis of asynchronous circuits

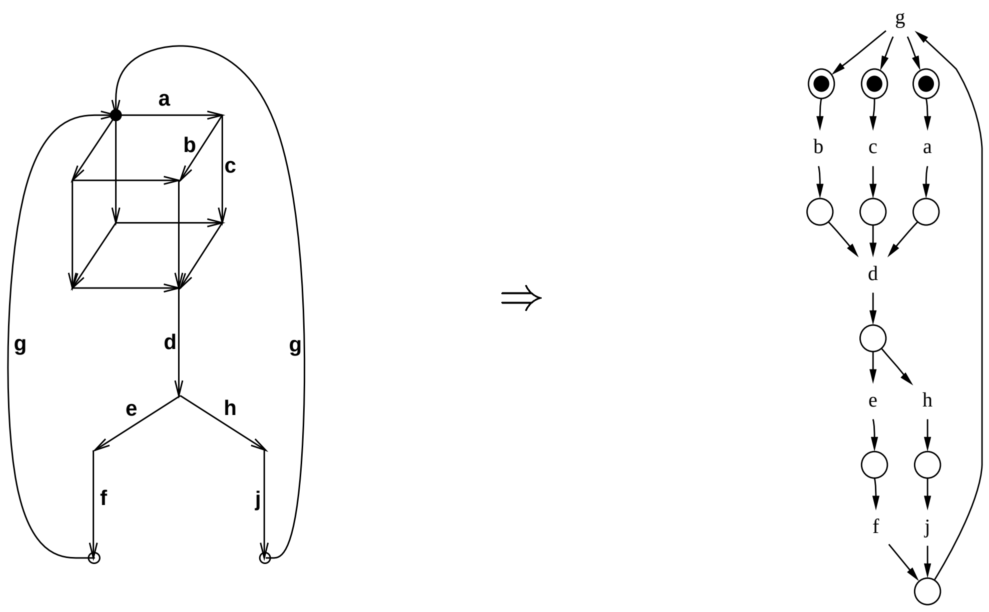


Figure 2: Synthesis of a PN

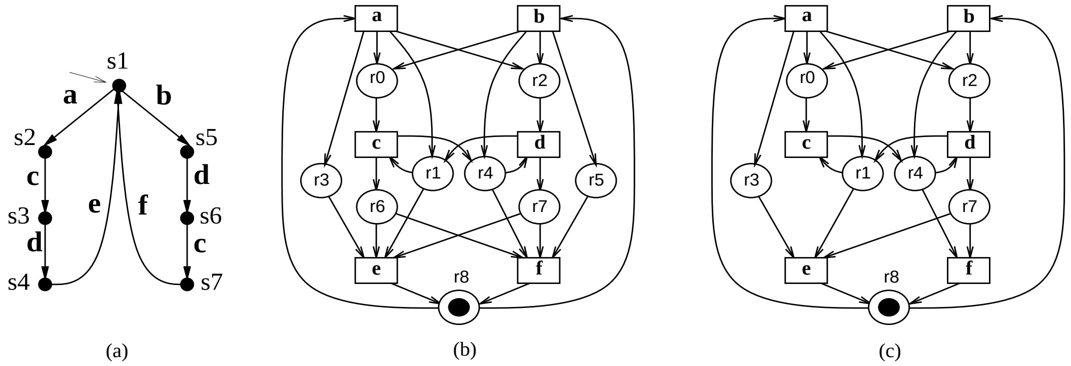


Figure 3: (a) Transition system. (b) Minimal saturated and (c) place-irredundant nets.

A user view of the circuit synthesis is illustrated by the example shown in Fig. 4. Given an initial STG specification (the left part of the figure), the tool realizes that the immediate construction of a net-list is not possible. Indeed, the property of Complete State Coding is not satisfied: different states of the system are encoded with the same binary code although they imply contradictory next values for at least one of the output signals. To resolve this state conflict `petrify` automatically inserts a new state signal ($csc0$). Transitions of this state signal ($csc0-$ and $csc0+$) are inserted in such way that the resulting logic is optimized according to a selected cost-function.

After inserting this state signal no state conflicts exists in the system and a speed-independent circuit can be constructed with C-elements [2] and complex gates (in the middle). However, these complex gates may not be available in the gate library. Assume, for example, that the library contains only simple gates with 2 inputs and C-elements. In such case, `petrify` will automatically perform combinational and sequential decomposition of the logic, preserving speed-independent

[2]C-element is an asynchronous latch with a next function $c = ab + ca + cb$, where a, b are inputs to the latch and c is its output.

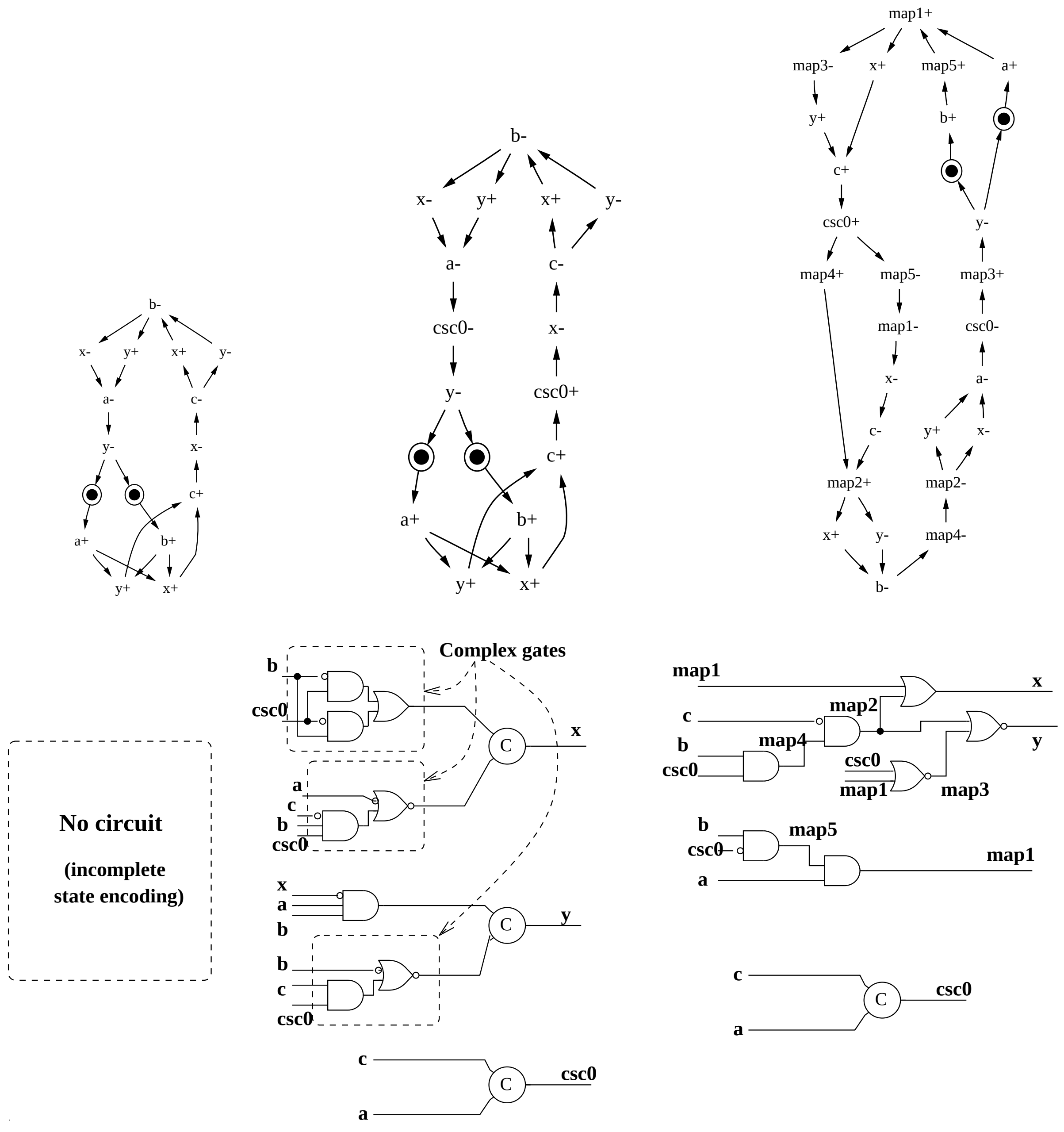


Figure 4: State encoding and technology mapping

properties and striving to minimize the logic. The final logic net-list for this library and the corresponding Signal Transition Graph will be automatically derived by the tool (Fig. 4, the right part).

3 Theory behind `Petrify`

The theory behind `petrify` is presented in [11, 12]. `Petrify` strives to *minimize* the number of places, in order to make the final Petri Net more understandable by the designer. It either generates a complete set of *minimal* regions (which are analogous to prime implicants in Boolean minimization) or further removes redundant regions (which is similar to generating a prime irredundant cover in Boolean minimization).

In the initial TS, all transitions with the same label are considered as one event. `Petrify` solves the problem of *merging* and *splitting* "equivalent" labels, i.e., those labels which model the same event, but must be split in order to yield a valid Petri Net. Therefore, the synthesis method is not limited to *elementary* TSs, which are quite restricted; we can handle the full class of TSs by means of label splitting. In the following sections we will briefly and informally review the theory behind `petrify`.

3.1 Basic models: Petri Nets and Transition Systems

Informally, a TS ([29]) can be represented as an arc-labeled directed graph. A simple example of a TS is shown in Fig.2 (the left part). A TS is called *deterministic* if for each state s and each label a there can be at most one state s' such that $s \xrightarrow{a} s'$. A TS is called *commutative* if whenever two actions can be executed from some state in any order, then their execution always leads to the same state, regardless of the order. For the purpose of synthesis of asynchronous circuits we are mainly interested only in deterministic and commutative TSs.

A Petri Net is a quadruple $N = (P, T, F, m_0)$, where P is a *finite* set of places, T is a *finite* set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and m_0 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 (denoted $m_1 \xrightarrow{t} m_2$). The right half of Fig.2 presents a PN expressing the same behavior as the TS shown in the left half of the same figure. Tokens represent the initial marking which corresponds to the top left state of the TS.

The set of all markings reachable in N from the initial marking m_0 is called its Reachability Set. The graph with vertices corresponding to the markings of a PN and with arcs connecting markings reachable in one transition is called the Reachability Graph (RG) of the PN.

A Signal Transition Graph (STG, [6, 34]) is a Petri net with transitions labeled with up and down transitions of signals (denoted by x^+ and x^- for signal x).

A PN is called

- *safe* if no more than one token can appear in a place in any reachable marking,
- *free-choice* if for each place p with more than one output transition each of this transitions has exactly one input place – place p, i.e., the enabling condition of conflicting transitions depends only on the marking of a single place.
- *place-irredundant* if removing any place from the PN will change the set of possible sequences of firing transitions (i.e., behavior of the net will be disturbed).

A PN in Fig.2 is safe, free-choice and place-irredundant.

3.2 Regions and Excitation Regions

Let S_1 be a subset of the states of a TS, $S_1 \subseteq S$. If $s \notin S_1$ and $s' \in S_1$, then we say that transition $s \xrightarrow{a} s'$ *enters* S_1. If $s \in S_1$ and $s' \notin S_1$, then transition $s \xrightarrow{a} s'$ *exits* S_1. Otherwise, transition $s \xrightarrow{a} s'$ *does not cross* S_1. A *region* 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 TS shown in Fig.3(a). The set of states $r_2 = \{s_2, s_3, s_5\}$ is a region, since all transitions labeled with a and with b enter r_2, and all transitions labeled with d exit r_2. Transitions labeled with c do not cross r_2. On the other hand, $\{s_2, s_3\}$ is not a region since transition $s_3 \xrightarrow{d} s_4$ enters this set, while another transition also labeled with d, $s_5 \xrightarrow{d} s_6$, does not.

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. The set of all pre-regions and post-regions of e is denoted with ${}^\circ e$ and e° respectively.

While regions in a TS are related to places in the corresponding PN, an excitation region for event a is a maximal set of states in which transition a is enabled. Therefore, excitation regions are related to transitions of the PN. More formally, a set of states is called a *generalized excitation region* (denoted by $GER(a)$) for event a if it is a *maximal* set of states (a set of states with a given property is maximal if it is not a subset of any other set with this property) such that for every state $s \in GER(a)$ there is a transition $s \xrightarrow{a}$. Sometimes it is more convenient to consider connected subsets of GERs. A set of states is called an *excitation region* (denoted by $ER_j(a)$) if it is a *maximal connected* set of states such that for every state $s \in ER_j(a)$ there is a transition $s \xrightarrow{a}$. Since any event a can have several separated ERs, an index j is used to distinguish between *different connected occurrences* of a in the TS. In the TS from Fig.3(a) there are two excitation regions for event d: $ER_1(d) = \{s_3\}$ and $ER_2(d) = \{s_5\}$, while $GER(d) = \{s_3, s_5\}$.

3.3 Deriving PNs based on the excitation closure

Given a set of all minimal regions (a region is called minimal if it is not a superset of any other region) let us build a PN following four rules:

- For each event e of the TS a transition labeled with e is generated in the PN;
- For each minimal region r a place r is generated;
- Place r contains a token in the initial marking iff the corresponding region r contains the initial state of the TS;
- The flow relation of the PN is as follows: transition labeled with e is an output transition for place r iff r is a pre-region of event e in the TS and e is an input transition of r iff region r is a post-region of e.

As shown in [11], if the following two conditions hold then a PN derived by the four rules above is bisimilar to the original TS. Bisimilar [25] means that behavior of the TS and the PN cannot be distinguished by the external observer who can only see the events of these two models.

- *Excitation closure:* For each event e the intersection of pre-regions is equal to its generalized excitation region.
- *Event effectiveness:* For each event e there is at least one pre-region.

Moreover, one may remove regions still preserving behavior of the PN until excitation closure is violated. By removing regions from the set of all minimal regions while still keeping the excitation closure condition `petrify` generates a place-irredundant PN. By further merging of the minimal regions a place-minimal net can be generated.

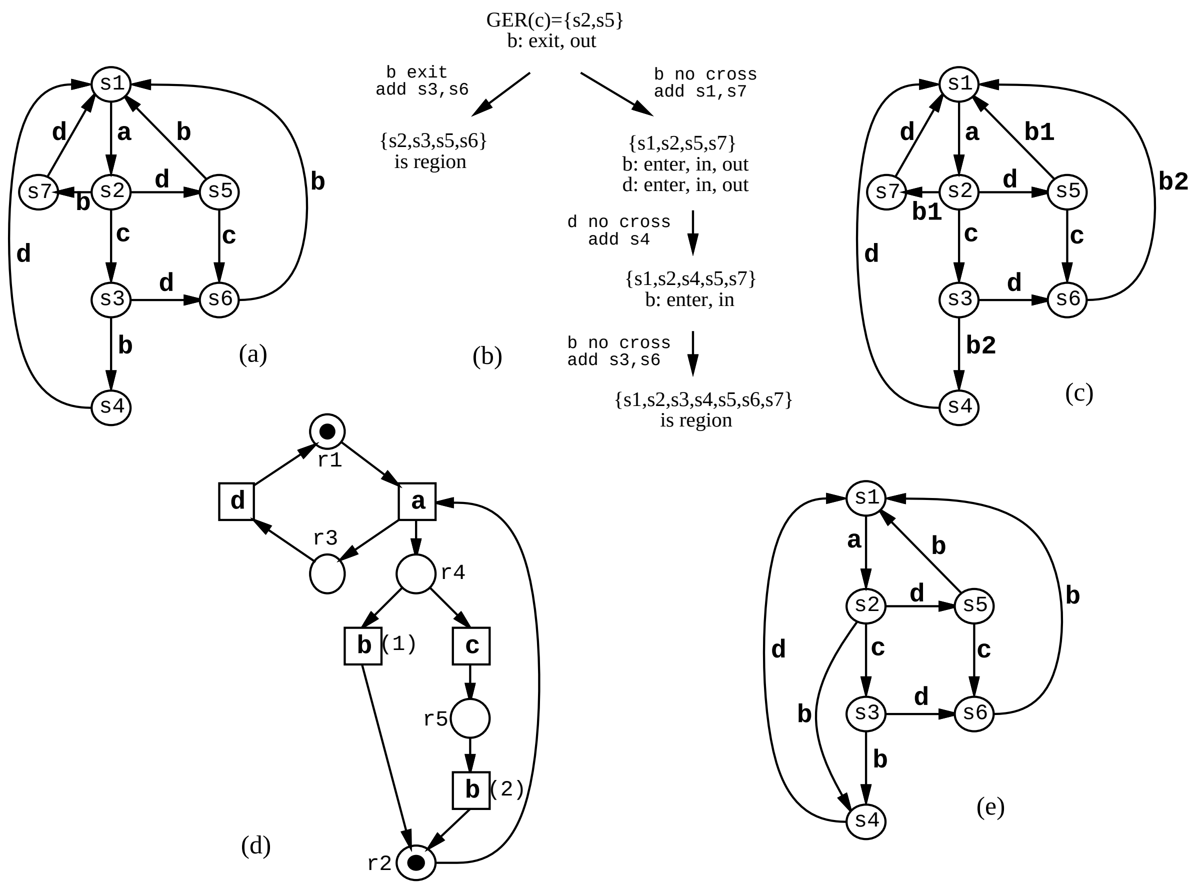


Figure 5: (a) TS, (b) expansion tree for pre-regions of event c, (c) excitation closed TS, (d) PN, (e) reachability graph of the PN

3.4 Generating minimal regions and label splitting

The set of minimal pre-regions of an event e is calculated by gradually expanding its generalized excitation region to obtain sets of states that do not violate the "entry-exit" relationship. When the excitation closure is not fulfilled, i.e.

$$\bigcap_{r \in {}^{\circ}a} r \neq GER(e)$$

some events must be split to satisfy this condition.

The strategy to split events is explained by the example shown in Fig.5 for the pre-regions of event c. Initially, $GER(c) = \{s_2, s_5\}$ is taken for expansion. Event b violates the region conditions, since two transitions labeled with b exit $\{s_2, s_5\}$ and two other transitions labeled with b are outside $\{s_2, s_5\}$. Next, two possible legalizations for event b are considered:

- Two input states for transitions of b, which are not yet included into the constructed set of states, $s3$ and $s6$, are added into the set. Now event b exits set $\{s_2, s_3, s_5, s_6\}$. Since no other violations of region conditions are found this set is a region.
- Two output states for transitions of b, $\{s_1, s_7\}$, which are not yet included into the set are added to the set in the attempt to make b non-crossing. This attempt fails since more

violations of the region conditions are found and further expansions are applied until all branches of the search tree find a region.

The example illustrates how all branches will eventually be pruned, in the worst case, when covering the whole set of states.

Let us call r' the intersection of the regions found in the expansion. We have

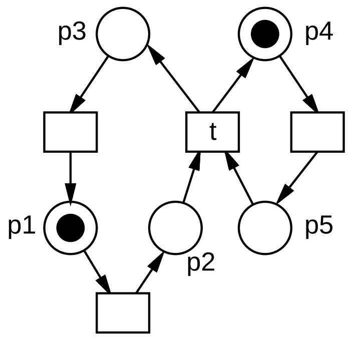
$$r' = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\} \cap \{s_2, s_3, s_5, s_6\} = \{s_2, s_3, s_5, s_6\}$$

The strategy for label splitting will take all those explored sets r such that

$$\{s_2, s_5\} \subseteq r \subset r'$$

All three states explored before finding regions are good candidates. However, the set $\{s_2, s_5\}$ is the best one by the fact that only one event violates the crossing conditions and it makes the intersection of pre-regions smaller (closer to GER). Thus, event b is split into two new events (b_1 and b_2) for $\{s_2, s_5\}$ to become a region. The new TS is equivalent to the original (up to renaming of the split events). The corresponding PN is shown in Fig.5(d) and its RG in Fig.5(e). Note that it contains one state less than the original TS, due to the implicit minimization for equivalent states $s4$ and $s7$ (states $s4$ and $s7$ are equivalent since there is only one output transition for each of them, labeled with b, and each of these transitions enter state $s1$).

3.5 Internal representation of the objects



- marking: $p_1\bar{p}_2\bar{p}_3p_4\bar{p}_5$
- region, set of states: p_1, p_2p_5
- flow relation (for t): $(p_2p_5\bar{p}_3\bar{p}_4) \cdot (\bar{q}_2\bar{q}_5q_3q_4) \cdot (p_1 \Leftrightarrow q_1)$

Figure 6: Symbolic representation of Petri net objects in `petrify`

The proposed method requires a broad exploration of sets of states of a TS. Moreover, operations such as intersection, inclusion and equality among the explored sets must be executed often. An efficient representation of the TS and its states is thus crucial to cope with the complexity of such operations.

Given an appropriate encoding of the states of the TS, we have chosen to use Ordered Binary Decision Diagrams [4] to represent sets of states (by means of characteristic functions) and the TS (by means of the disjunction of transition relations, one for each label). The algorithms to manipulate the sets of states of the TS are based on symbolic techniques for verification of sequential machines [13].

For deriving a TS from the initial PN `petrify` performs a token flow analysis of the initial STG and produces a transition system in symbolic form, using Binary Decision Diagrams. The latter represent boolean characteristic functions of markings, states, sets of states and the flow relation as shown in Fig.6.

4 Theory behind asynchronous circuit synthesis

4.1 Asynchronous circuits and speed-independence

An asynchronous circuit is an arbitrary interconnection of logic gates such that no two gate outputs are connected together ([36, 27]). Each logic gate is characterized by a Boolean equation describing the gate output as a function of the gate inputs and (if the gate is *sequential*, rather than *combinational*) of the gate output.

The behavior of a circuit can be completely characterized by using a TS with one state for each Boolean vector representing the values of the gate outputs and of the primary inputs of the circuit (collectively called *signals*). An example of an asynchronous circuit is given in Fig. 4.

Roughly speaking, a circuit is defined to be speed-independent if its behavior remains correct under any changes of gate delays. No hazards are possible in speed-independent circuits under any input changes (possibly multiple input changes in non-fundamental mode) [23, 20].

4.2 Property-preserving event insertion

Event insertion is informally seen as an operation on a TS which selects a subset of states, splits each state in it into two states and creates, on the basis of these new states, an excitation and switching region for a new event. Fig. 7 shows the chosen insertion scheme, analogous to that used by most authors in the area, in the three main cases of insertion with respect to the position of the states in the insertion set, denoted $ER(x)$ (*entrance* to, *exit* from or *inside* $ER(x)$).

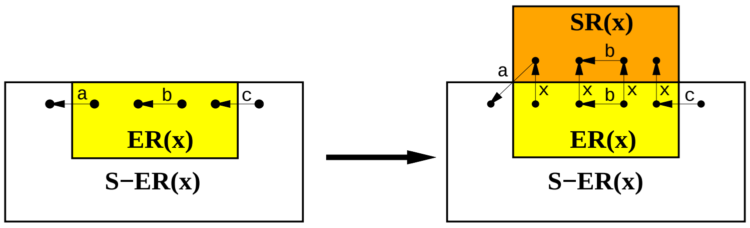


Figure 7: Event insertion scheme

State signal insertion must also preserve the *speed-independence* of the original specification, that is required for the existence of a hazard-free asynchronous circuit implementation.

Let a TS has a set of events E and a set of transitions T. An event a of the TS is said to be *persistent in a subset S' of states* of S iff $\forall s1 \in S', b \in E : [s1 \xrightarrow{a} \wedge (s1 \xrightarrow{b} s2) \in T] \Rightarrow s2 \xrightarrow{a}$. An event is said to be *persistent* if it is persistent in S. For a binary encoded TS, determinism, commutativity and output event persistency guarantee speed-independence of its circuit implementation. Insertion sets should be chosen in such a way that persistency and commutativity of the original events are not violated.

The following property of insertion sets, based on theory developed in [8], provides a rationale for our approach.

Property 4.1 *Regions, excitation regions and intersections of pre-regions can be used as insertion sets in a commutative and deterministic* TS.

This property suggests that the 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 nice sets of states can be built very efficiently, from "bricks" (regions) rather than "sand" (states).

4.3 Selecting excitation regions for new signals

Assume that the set of states S in a TS is partitioned into two subsets which are to be encoded by means of an additional signal. This new signal can be added either in order to satisfy the CSC condition, or to break up a complex gate into a set of smaller gates. In the latter case, a new signal is added to represent the output of the intermediate gates added to the circuit and the speed-independent implementability of the decomposed specification is checked again ([5]).

Let r and $\overline{r} = S - r$ denote the blocks of such a partition. In order to implement such an encoding, we need to insert appropriate transitions of the new signals in the *border states* between the two subsets.

`Petrify` considers the so-called *exit border* (EB) of a partition block r, denoted by $EB(r)$, which is informally a subset of states of r with transitions exiting r. We call $EB(r)$ *well-formed* if there are no transitions leading from states in $EB(r)$ to states in $r - EB(r)$. Symmetrically, *input borders* can be handled. Fig. 8 illustrates the notions of exit and input borders.

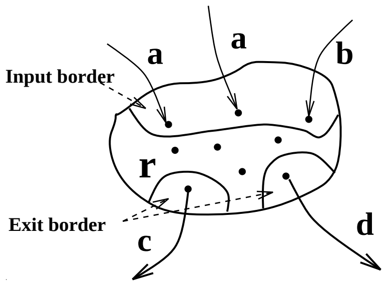


Figure 8: Exit and input borders

Note that we need each new signal x to orderly cycle through states in which it has value 0, 0*, 1 and 1*. We can formalize this requirement with the notion of *I-partition* ([37] used a similar definition).

An I-partition divides a set of all states of a TS into four blocks: S^0, S^1, S^+ and S^-. $S^0(S^1)$ defines the states in which x will have the value 0 (1). $S^+(S^-)$ defines $\mathsf{GER}(x+)$ ($\mathsf{GER}(x-)$). For a consistent encoding of x, the only allowed events crossing boundaries of the blocks are the following: $S^0 \to S^+ \to S^1 \to S^- \to S^0$, $S^+ \to S^-$ and $S^- \to S^+$ (the latter two would cause a persistency violation, though). The problem of finding an I-partition is reduced to finding a bipartition S and is done in four steps:

1. Find a bipartition of states $\{b, \overline{b}\}$
2. Calculate $EB(b)$ and $EB(\overline{b})$ (similarly for input borders)
3. Extend EBs to well-formed EBs by backward closure
4. Check that persistency condition is not violated

Three first steps are shown in Fig. 9.

4.4 Gate-level speed-independence conditions

Necessary and sufficient conditions for speed-independent implementation using unbounded fanin *and* gates (with unlimited input inversions), bounded fanin *or* gates and C elements were given in [21] (extending a previous result of [2]). `Petrify` uses a basic implementation architecture, called

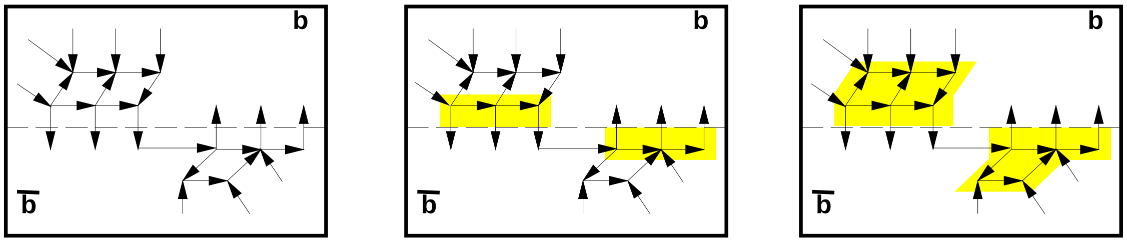


Figure 9: From bipartition to I-partition

the *standard-C* architecture (Fig. 10). Contrary to the previous tools instead of unbounded fanin gates for the first level, `petrify` can search for *implementable* gates, that is gates which exist in the chosen library.

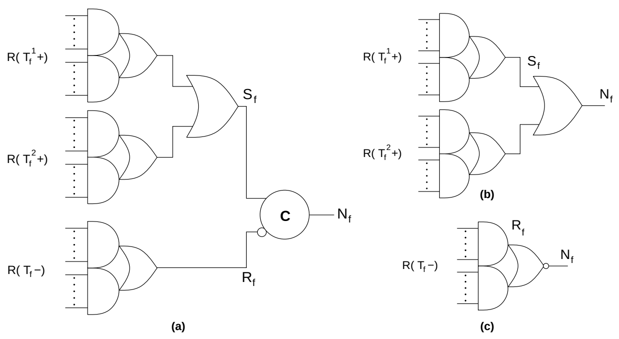


Figure 10: The standard-C architecture extended for complex gates

The basic idea of the standard-C implementation architecture is that every first-level gate implements an up or down transition of the user-specified signal behavior. In order to ensure speed-independent operation, a number of constraints that are collectively called the *monotonous poly-term cover conditions* ([21]) must be satisfied.

In the following we will consider partitions of the set of excitation regions of a given signal a into *joint excitation regions* $ER_j(a^*)$. The word "joint" here indicates that a few excitation regions can be joined together and implemented with one logic gate in the circuit.

The *joint quiescent region* $QR_j(a^*)$ of a given signal transition with joint excitation region $ER_j(a^*)$ is a *maximal* set of states s such that:

- a is stable in s, and
- s is reachable from $ER_j(a^*)$ only through states in which a is stable, and
- s is not reachable from any other $ER_k(a^*)$ such that $k \neq j$ without going through $ER_j(a^*)$.

Similarly, the *backward region* $BR_j(a^*)$ is a maximal set of states s such that:

- a is stable in s, and
- $ER_j(a^*)$ is reachable from s only through states in which a is stable, and
- no other $ER_k(a^*)$ such that $k \neq j$ is reachable from s without going through $ER_j(a^*)$.

Let $C_j(a^*)$ denote one of the first-level gates in the standard-C architecture. $C_j(a^*)$ is a correct monotonous poly-term cover for the joint excitation region $ER_j(a^*)$ if:

1. $C_j(a^*)$ covers (i.e., its Boolean equation evaluates to 1) all states of $ER_j(a^*)$.
2. $C_j(a^*)$ covers only states of $ER_j(a^*) \cup QR_j(a^*) \cup BR_j(a^*)$.
3. If $C_j(a^*)$ covers some state s of $BR_j(a^*)$, then s is also covered by some other $C_k(a^*)$ such that a_j^* and a_k^* are complementary (up and down or down and up, respectively) and $s \in BR_j(a^*) \cap QR_k(a^*)$.
4. $C_j(a^*)$ has exactly one up and one down transition in any sequence of states within $ER_j(a^*) \cup QR_j(a^*) \cup BR_j(a^*)$.

Under these conditions, it is possible to show that the outputs of the first-level gates are *one-hot encoded*, and that means that any valid Boolean decomposition of the second-level *or* gates will be speed-independent.

The chosen architecture is general enough to cover the case in which a signal in the specification admits a *combinational* implementation, because in that case the set and reset network are the complement of each other, and the C element with identical inputs can be simplified to a wire.

4.5 Strategy for technology mapping

The strategy for technology mapping which is implemented in the procedure for selecting the best I-partitions and in the cost function is based on two iterative steps:

- Combinational decomposition and extraction of set and reset functions
- If no valid combinational decomposition can be found, then additional state signals are inserted preserving speed-independence to increase the don't care set and to simplify the logic.

Special conditions for correct speed-independent decomposition must be preserved, since each signal transition at the decomposed gate must be acknowledged by some other gate in the speed-independent circuit. Contrary to conditions from [5] `petrify` allows gate sharing and fit well in our region-based partitioning of the states. The simple gate circuit shown in Fig. 4 is obtained from the complex gate circuit by combinational decomposition. Note that some of the C-elements were eliminated.

5 Conclusions

Petri nets have shown to be an appropriate formalism to describe the behavior of systems with concurrency, causality and conflicts between events. For this type of systems, the method presented in this paper allows to transform different models (CSP, CCS, FSMs, PNs) into a unique formalism for which synthesis, analysis, composition and verification tools can be built.

Synthesizing Petri nets from state-based models is a task of reverse engineering that abstracts the temporal dimension from a flat description of the sequences of events produced by the system. The synthesis method discovers the actual temporal relations between the events. The symbiosis among the notions of TS, *region* and *excitation region* in the same method has been crucial to derive efficient algorithms both for manipulating concurrent specifications and algorithms for synthesis and optimization of asynchronous circuits.

For the future directions we consider extending `petrify` for handling:

- unsafe, general PNs;
- synthesis of synchronous parallel controllers;
- applications to a hardware/software codesign of reactive controllers.

How to get and use Petrify

You can get the tool from the following www address: http://www.ac.upc.es/~vlsi/petrify/petrify.html.

There is a man page there describing the syntax for representing input PNs, STGs and TSs and possible options for `petrify`.

References

[1] E. Badouel, L. Bernardinello, and Ph. Darondeau. Polynomial algorithms for the synthesis of bounded nets. Technical Report 2316, INRIA, RENNES Cedex, France, 1994.

[2] P. A. Beerel and T. H-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[3] L. Bernardinello, G. De Michelis, K. Petruni, and S. Vigna. On synchronic structure of transition systems. Technical report, Universita di Milano, Milano, 1994.

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

[5] 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.

[6] T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.

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

[8] 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*, pages 36–47, March 1996.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Coupling technology mapping, logic optimization and state encoding. Technical report, Universitat Politecnica de Catalunya, 1996.

[10] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *Proceedings of the Design Automation Conference*, June 1996. to appear.

[11] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. In *Proc. of ICCAD'95*, pages 164–171, November 1995.

[12] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. Technical Report RR 95/09 UPC/DAC, Universitat Politecnica de Catalunya, April 1995.

[13] 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.

[14] J. Desel and W. Reisig. The synthesis problem of Petri nets. Technical Report TUM-I9231, Technische Universität München, September 1992.

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

[16] D. Drusinsky. Extended state diagrams and reactive systems. *Dr.Dobb's Journal*, pages 72–80,106–107, October 1994.

[17] A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I, II. *Acta Informatica*, 27:315–368, 1990.

[18] C. A. R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, pages 666–677, August 1978.

[19] Henrik Hulgaard and Steven M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.

[20] 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.

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

[22] R. P. Kurshan. Analysis of discrete event coordination. In *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[23] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic Publishers, 1993.

[24] Robin Milner. A calculus of communication systems. In *Lecture Notes in Computer Science*, volume 92. Springer-Verlag, 1980.

[25] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[26] M. Mukund. Petri nets and step transition systems. *Int. Journal of Foundations of Computer Science*, 3(4):443–478, 1992.

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

[28] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

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

[30] S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

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

[32] M. Pezzé, R. N. Taylor, and M. Young. Graph models for reachability analysis of concurrent programs. *ACM Transactions on Software Engineering and Methodology*, 4(2):171–213, 1995.

[33] T. G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[34] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets, Torino, Italy*, 1985.

[35] D.C. Tsichritzis and P.A. Bernstein. *Operating Systems.* Academic Press, London, 1974.

[36] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[37] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on Signal Transition Graphs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 112–117, November 1992.