Performance Analysis and Behavior of Timed Concurrent Systems Using Petri-Net Models

Steven M. Nowick

Columbia University, New York NY 10027, USA,
nowick@cs.columbia.edu,
WWW home page: http://www.cs.columbia.edu/~nowick/

Abstract. The modeling and performance analysis of concurrent systems using Petri nets is considered under a useful timing model: probabilistic (i.e. exponential) delay distributions. While the focus is on decision-free concurrent systems, i.e. marked graphs (MG), generalizations are identified to allow limited choice. A tight state space is efficiently constructed, through hierarchical decomposition followed by composition, which defines the time evolution of the system. Efficient algorithmic solution techniques and tools are also presented. Interesting analogies to biological and physical systems are highlighted.

Keywords: asynchronous circuits, Petri nets, marked graphs, timing analysis, stochastic performance analysis

1 Introduction

Asynchronous design have been widely explored as a promising direction for organizing complex systems [1] [2]. While much of the asynchronous paradigm has been directed towards digital systems, the underlying paradigm of "self-synchronizing systems" which evolve in continuous time, has resonance in the physical, biological and social worlds.

A key research direction has been to capture complex system behavior, then analyze its behavior. The main focus of this article is on asynchronous digital systems. However, we also outline potential synergy of these models and analytical techniques to other self-organizing systems, which evolve over time with loosely-coupled concurrent behavior, from an initial state to a steady-state periodic behavior.

Petri nets have been widely used for the analysis, synthesis and optimization of asynchronous circuits and systems [3] [4]. These will be used as the foundational models for the approach outlined in this article.

2 Motivation and Overview

There are several major technical difficulties involved in the performance analysis of asynchronous systems.

Unlike clocked systems where clock boundaries form natural partitions for logic between stages to be analyzed individually, an asynchronous system is inherently nonlinear, meaning there is no easy way to partition the system into independent subsystems. The system has to be analyzed as a whole. This has often led to unmanageable state space problems.

In addition, as the system is event-driven, arbitrary arrival time of inputs and variations in data-dependent delays in individual components can have significant impact on the overall performance of the system, and must therefore be taken into account during performance analysis. As pointed out in [5], taking the statistical average of the processing time of individual components is often inadequate in determining the average performance of the overall system; the so-called "variance" of processing times must also be considered.

Finally, there is no clear consensus in the asynchronous design community on what performance metrics are useful for characterizing the performance of a system and for identifying bottlenecks for optimization.

This article addresses these issues by providing an efficient and general method for analyzing the asymptotic performance of asynchronous systems. In our approach, an asynchronous system is modeled as a marked graph, a subclass of Petri nets that captures concurrency and data-dependent relationships between interacting components in decision-free systems [6].[1] The variations in input arrival time and component delays are captured in a probabilistic delay model. The probability distribution of input arrival at a component can provide indication of system bottlenecks. Component utilization, as well as system latency and throughput, can be derived as measures of system performance.

This article focuses on the subclass of asynchronous systems that can be modeled by strongly-connected marked graphs, namely, decision-free systems. A sketch of how to extend our approach to systems with choice is presented in [7] [8]. The subclass of decision-free systems, while limited, has been shown to be capable of modeling many interesting concurrent systems [9]. More importantly, it forms a foundation to develop theories and algorithms for analyzing more complex systems, which we will consider in future work. We also focus on modeling at the system architecture level; we do not currently consider circuit-level modeling issues such as delay variations due to process variations.

There are some key differences between our method and previous work. First, this article shows that the state transitions of a system modeled by a strongly-connected marked graph exhibit an inherent periodic structure, and the system is analyzed as a *periodic* Markov chain. Previous approaches do not take into account the inherent periodic property of the system, and the system is analyzed as *aperiodic*. Theoretically, one cannot properly adopt an aperiodic Markov model to describe a periodic system, or the system would not converge. Second, by exploiting the periodicity of the system, we derive an algorithm to generate an exact, tight state space of the system, and exploit the regularity of its structure for efficient memory management and to reduce the complexity of

[1] These systems are also referred to as "deterministic" concurrent systems, or concurrent systems without conditional behavior (or without "choice"), in the literature.

computation. In contrast, previous approaches can generate a large number of unreachable (and unnecessary) states, resulting in excessive memory usage and runtime. Third, our approach is targeted to generating the probability distribution of input arrival time to components, as well as component utilization, as metrics for performance. The approach in [10] targets the time separation of events. Though it is also possible for our tool to generate metrics based on time separation of events, we believe input arrival time gives more directly useful information for subsequent system optimization.

The key contributions summarized in this article are as follows. First, we identify that the state transitions in decision-free asynchronous systems exhibit an inherent periodic structure.[2] To the best of our knowledge, this is the first time this property has been exploited in analyzing the performance of such systems. Second, we present an algorithm to construct the precise reachable state space of the system. Third, we propose the use of the probability distribution of input arrival time as a metric for performance optimization. Finally, we present a practical tool to demonstrate the feasibility our approach. Initial experimental results are promising, showing several orders of magnitude improvement in both runtime and the size of the state space over previously published results.

This article summarizes our approach to the efficient analysis of asynchronous circuits and systems using probabilistic delay models. More details can be found in [7] [8] [12].

3 Marked graphs

Marked graphs are used to model concurrency and data-dependent relationships between interacting components in a concurrent system. They specify the reachable states of the system and their causality.

A marked graph is a triple $N = (E, T, F)$ where E is the set of places, T the set of transitions, and $F \subseteq (E \times T) \cup (T \times E)$ the flow relation. In a marked graph, every place has at most one input and one output transition. A transition is enabled to fire whenever there is a token in each of its input places. An enabled transition fires by removing one token from each of its input places, and depositing a token in each of its output places. A *marking* is an assignment of tokens to the places in the graph. A marked graph is *safe* if in any marking reachable from an initial marking M_o, every place contains no more than one token. A marked graph is *live* if every transition is fireable, or can be made fireable through some sequence of firings from the initial marking M_o.

In the context of discrete-event systems, the marking of a marked graph corresponds to the *state* of the concurrent system, and the firing of a transition corresponds to the occurrence of an *event*.

[2] Both [9] and [11] also exploited the existence of repeating structures in marked graphs in their approaches. However, the repeating structure they used is related to the *events* in the graph structure, rather than to the *states* in the dynamic behavior of the model.

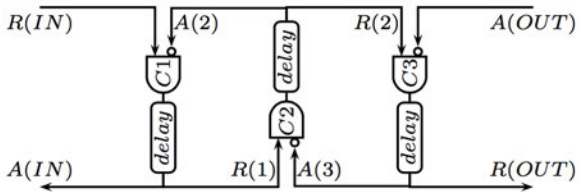

Fig. 1. Control circuit for a micropipeline.

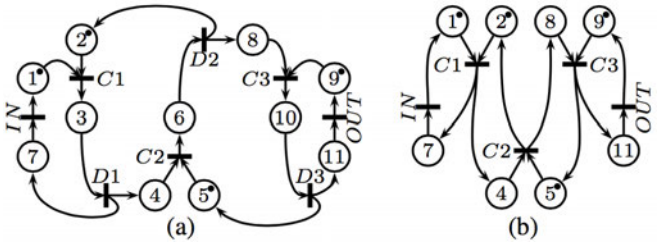

Fig. 2. Marked graph model for a micropipeline: (a) low level (b) abstract level.

Example: Figure 1 shows the the control circuit for a three-stage asynchronous micropipeline [13] proposed by Sutherland. Figures 2(a) and (b) show its corresponding marked graph models, representing two views: the former at a more detailed level with some low-level circuit components, and the latter at a more abstract level.

4 Periodicity of Strongly-Connected Marked Graphs

This section presents the first new research result: given a system which is modeled by a strongly-connected marked graph, the corresponding STG *always* exhibits a periodic behavior.

The following three theorems were proved by Commoner et al. [6] for marked graphs:

Theorem 1. *A given marking of a graph is live and safe iff every simple cycle has exactly one token, and through every place in the graph there is a simple cycle of token count one. A marking which is live remains live after firing.*

Theorem 2. *For every finite, directed, strongly-connected graph there exists a live, safe marking.*

Theorem 3. *Let M be a live marking of a strongly-connected marked graph, then for any firing sequence that leads back to the initial marking M, all i transitions have been fired an equal number of times. Furthermore, there exists a firing sequence leading from M to itself, in which every transition fires exactly once.*

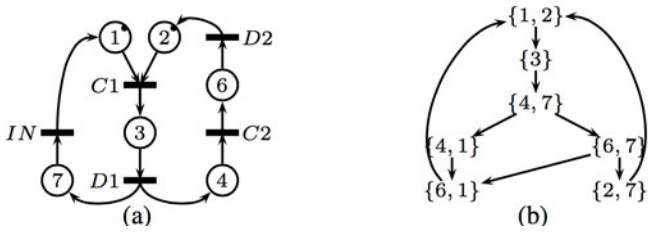

Fig. 3. A simple unit: (a) marked graph (MG) (b) signal transition graph (STG).

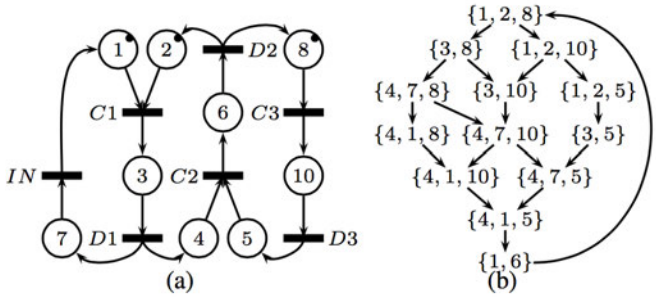

Fig. 4. Two coupled simple units: (a) marked graph (MG) (b) signal transition graph (STG).

Intuitively, Theorem 1 indicates that the marked graph can be viewed as a set of interacting simple cycles; a marking can only be live if there is at least one token circulating on each cycle, and can only be safe if each cycle has at most one token. The next two theorems address the existence of such a live safe marking, and the steady-state regular behavior of marked graphs with live markings.

The key result that is proved in this section is that any strongly-connected marked graph corresponds to an STG with periodic structure. This result is formalized in the following proposition:

Proposition 1. ***Periodic STG for strongly-connected MG***
Given any strongly-connected marked graph, its reachable set S of all live, safe markings (states) can be divided into δ disjoint sets $B_1, B_2, .., B_\delta$ such that for any state i, j in S, and if state i belongs to set B_k, i can make a transition to j iff j belongs to the immediate next set B_{k+1}. More formally, $P(i,j) = 0$ unless $i \in B_k$ and $j \in B_{k+1}$ for $k \in 1, 2, ..., \delta - 1$, or $i \in B_\delta$ and $j \in B_1$, where $P(i,j)$ is the probability of the system making a transition from state i to state j in one transition. In other words, after a state in set B_i is visited, it always takes δ firings for a state in the same set to be visited again. Such an STG is called a ***periodic*** *STG, and its* ***period*** *is δ.*

Figure 3 illustrates the intuitive idea behind Proposition 1. We define a *synchronization point* as a transition in a marked graph with more than one input

place. Figure 3(a) shows an example of a marked graph with a *single synchronization point,* which is defined as a **simple unit**. Figure 3(b) shows the corresponding STG of the marked graph. States in the same row of the STG form a set which can be reached through one firing only from states from the row above, and can reach through one firing only states from the row below. Intuitively, the synchronization point acts as a "gate" to constrain the movement of tokens in the marked graph, as it must wait for all its input tokens to arrive before it can fire. This gives a structural pattern to the transition of states.

Two corollaries follow immediately from Proposition 1. Corollary 1 shows that the periodicity of the state transitions in a strongly-connected marked graph is preserved through the composition of two such graphs. This property will be used in in the next section to derive an algorithm to construct the state space of the system.

Two marked graphs are said to be "composed together" if they are joined at one or more transitions. An example is shown in Figure 4(a), where two simple units are composed at transitions $C1$ and $C2$. The corresponding STG is shown in Figure 4(b).

Corollary 1. *Given two marked graphs with periodic STG's, the STG of their composition is also periodic.*

The next corollary shows that the state transitions in a system modeled by a strongly-connected marked graph can be represented in a canonical form. This result will be used later to efficiently solve for the stationary distribution of the resulting Markov transition matrix.

Corollary 2. *The transition matrix of the underlying Markov chain of a strongly-connected marked graph can always be expressed in the following canonical form, where δ is the period, and $B_1, B_2, ..., B_\delta$ are disjoint sets of states.*

$$P = \begin{pmatrix} 0 & B_1 & . & . & 0 \\ . & & . & & . \\ . & & & . & . \\ 0 & & & & B_{\delta-1} \\ B_\delta & 0 & . & . & 0 \end{pmatrix}$$

5 Constructing the STG: an Algorithm

Now that it has been shown that an STG derived from any strongly-connected marked graph exhibits a periodic structure, this section presents an algorithm for constructing the periodic STG itself. The algorithm has two operators: *decomposition* and *composition*. The decomposition operator decomposes a marked graph into simple units with single synchronization points. The algorithm then constructs an STG for each of the simple units. Note that, from Proposition 1, the STG's of these simple units are periodic. The composition operator combines

```
decompose(marked_graph)
  foreach sync_pt in marked graph
    // find all simple cycles that goes through sync_pt
    foreach output_place at sync_pt
      find simple path that leads back to sync_pt
    // build state transition graph
    i = 0;  // counts period
    s[0][0] = concat(output places at sync_pt)
    do until (next_transition == sync_pt)
      i++;
      j=0;
      foreach enabled_transition in current_state
        fire;
        s[i][j] = new_state
        j++;
```

Fig. 5. decompose

```
compose(stg1, stg2)
  foreach state s1 in stg1
    foreach state s2 in stg2
      if s1 and s2 shares common places
        combine into new state and put in new_stg
      else if s1 and s2 do not share common places
        put state into single_state_vec
  while (single_state_vec != empty) do
    foreach state ss in single_state_vec
      foreach state ns in new_stg
        if ss and nw share common places
          combine into new state and put in new_stg
  return new_stg
```

Fig. 6. compose

the periodic STG's from two intersecting simple units into a single STG. The new STG is again periodic according to Corollary 1.

The entire state space of a live, safe marked graph can thus be built by first applying the decomposition operator, and then the binary composition operator on all decomposed units recursively. Figures 5 and 6 show the pseudo-code for the decomposition and composition operators, respectively. The state space constructed using this algorithm is tight: it includes only states that are reachable from an initial live, safe marking, and no others. No extra reachability analysis steps such as that used in [10] is required. Furthermore, the constructed state space can immediately be put into the canonical form shown in Corollary 2.

The intuition behind the algorithm is as follows. Since the marked graph is strongly connected, it can be broken down into a finite number of simple cycles, each of which with a token circling around it, according to Theorem 1. Each of these simple cycles can then be viewed as a oscillator, with a "period" equal to the number of transitions in the cycle. The composition of two simple cycles is then analogous to the coupling of two oscillators. The two coupled oscillators form a new oscillatory system, with a new period determined by the number of transitions in the new system, according to Corollary 1. The number of transitions in the new system is in turn determined by how "strong" the coupling between the two oscillators are: the stronger the coupling (meaning the more transitions they share in common), the shorter is the period of the resulting coupled system.

Another interesting observation that can be made on the algorithm is that once the STG's of the decomposed simple units are constructed, the STG of the entire design space is composed by simply "stitching" the lower level STG's together recursively. In other words, the state space of the entire system is constructed based solely on the structure of marked graph. There is no notion of the dynamic behavior of the system during the construction, i.e., the actual marking plays no role in the analysis. In contrast, most existing algorithms for state space exploration requires keeping track of a set of current states and computing a set of legal next states using a search strategy. The complexity of our algorithm is considerably lower.

6 Obtaining Performance Analysis Results

The result of the previous algorithm is a periodic STG, which can be transformed into a periodic Markov transition matrix (see [7] [8]). From the transition matrix of a Markov chain, one can obtain the asymptotic behavior of a system by solving the matrix for its *stationary distribution*. Techniques for finding the stationary distribution of Markov chains can be found in standard textbooks on stochastic processes.

Let P be the transition matrix of an irreducible Markov chain with recurrent periodic states of period δ. Then the transition matrix $\bar{P} = P^{\delta}$ can expressed as:

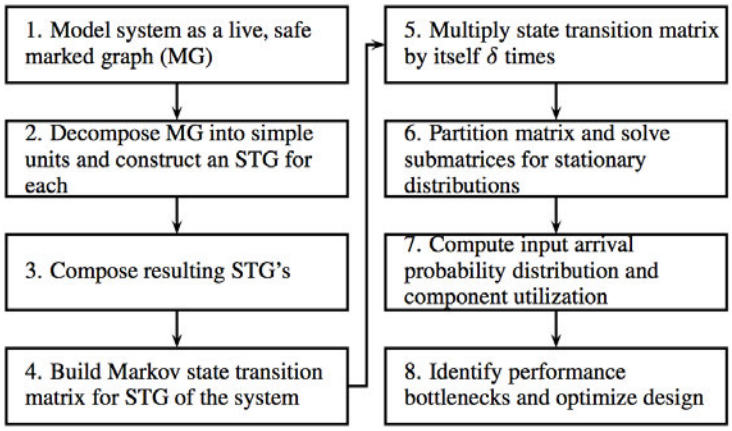

Fig. 7. Tool flow summary.

$$\bar{P} = \begin{pmatrix} P_1 & & 0 \\ & P_2 & \\ & \ddots & \\ & & \ddots \\ 0 & & P_\delta \end{pmatrix}$$

The sub-matrices $P_1, P_2, ...P_\delta$ form δ closed sets, each of which is irreducible, recurrent and aperiodic. The stationary distribution $\pi_1, \pi_2, ..., \pi_\delta$ for each of the sub-matrices can be solved.

Figure 7 shows a summary of the performance analysis flow.

Once the stationary distribution for each sub-matrix is found, useful performance metrics can be obtained as guides for optimization. For example, the stationary state distribution gives direct information on *input arrival time*, i.e., the probability of each input to a component arriving last (or first). System bottlenecks can be identified as paths in the system that lead to a late input arrivals with a high probability. *Component utilization* can be obtained by computing the probability of the component being in a state waiting for one or more of its inputs to arrive, divided by the probability of it being in a state where all input tokens are present. *System latency*, i.e., the average time separation between a given input event and output event of the system, can be found by computing the weighted average of all paths of executions of the corresponding states in the STG. Similarly, *system throughput* can be found by computing the average time separation between two events of the same output in successive iterations of the system.

7 Public-Domain Tool: DES Analyzer

We have developed a public-domain asynchronous performance and timing analysis package, *DES (Discrete Event System) Analyzer* (for download access and tutorials, see [12]).

It includes two tools: (i) *DES-PERF*, which uses user-supplied stochastic information to compute asymptotic system performance, and (ii) *DES-TSE*, which uses user-supplied min/max delay bounds on individual events to compute the global min/max "time separation of events" between any two pairs of events.

The algorithm flow in this article is implemented in the former tool, DES-PERF. (The latter tool explores an alternative timing model, using min/max delays on each marked graph edge.)

The package also contains a detailed tutorial, test examples, and overview of theory.

8 Experimental Results

Table 1 shows a summary of results for a number test cases ran on an Intel Xeon CPU at 3.06GHz with 1GB of memory.

test case	design instance	# reachable states	CPU Time (seconds)
micropipeline [13]	3-stage	29	0.003
	4-stage	70	0.007
	5-stage	169	0.031
	6-stage	408	0.164
	7-stage	985	0.986
	8-stage	2377	7.820
	9-stage	5740	29.835
	10-stage	13859	361.621
	11-stage	33460	4686.126
Huffman decoder [14]		160	0.036
DiffEq [15]		175	0.039
DCT [16]		169	0.031

Table 1. Experimental Results

The first set of test cases are different variants of Sutherland's micropipeline design [13] shown in Figure 1. The second test case is an asynchronous Huffman decoder design proposed in [14]. The third test case is a low-control-overhead asynchronous differential equation solver proposed in [15], and the fourth is an asynchronous DCT matrix-vector multiplier proposed in [16]. In each testcase, the architectural flow diagram is converted to a marked graph to capture data dependency between functional unit and concurrency.

In [10], results were reported for a 6-stage pipeline design similar to our test case 1, and reported reachable states of up to 28,000 and runtime of just less than one hour on a SPARC 20. As a followup, in [11], it was indicated that the

approach in [10] cannot handle pipeline designs of more than 8 stages. Our result shows significant improvements over their reported results. The performance results for an 8-stage pipeline can be obtained in 7.8 seconds. While different computing environment precludes any meaningful comparison in runtime, we would like to point out that, first, our state space is much smaller, as our algorithm prunes all unreachable and transient states from analysis. Second, the matrices we solve are also smaller, as their average size is the total number of states divided by the period of the system. Third, a stationary distribution for each of the sub-matrices in our analysis is guaranteed to exist. In fact, without our pruning of unreachable and transient states, and without taking into account the periodicity of the system, a stationary distribution for the system theoretically does not exist.

A few other notes can be made on the results. First, the size of the state space is exponential in the number of parallel operations that can run in the system at the same time. When applied to real designs, this result is not as pessimistic as it seems, as the number of process running in parallel is often limited. A full-buffer pipeline where each stage performs a different operation represent the worst case scenario, while many highly-concurrent systems have more moderate amounts of concurrency.

Secondly, while prior work [10] reported the solving for the stationary distribution of the transition matrix of the Markov chain as the bottleneck of the analysis, the same operation does not present a problem in our analysis. In fact, profiling shows the runtime is dominated by the matrix multiplication in Step 5 of the tool flow. Matrix multiplication is an $O(n^3)$ operation in general. Due to the regular structure of the matrix in our application, the multiplication can be simplified, and the complexity is $O(m^3)$, where m is the size of the largest sub-matrix of the system. The memory requirement for storing the matrix is greatly reduced as well, as only the submatrices need to be stored.

9 Conclusions, Observations and Future Work

We have presented an efficient solution for in analyzing the asymptotic performance of asynchronous systems.

We model an asynchronous system as a marked graph, and capture its underlying state transition as a Markov chain. We showed that the state transition graph of a system modeled by a marked graph exhibits a periodic structure, and proposed an algorithm for constructing a tight state space of the system based on this property, which is then transformed to a Markov chain. Local asymptotic performance metrics, such as *distribution of input arrival time* and *component utilization*, are obtained by solving the transition matrix of the Markov chain for its stationary state distribution. Using this information, global metrics for system bottleneck and cycle slack can then be derived, which can in turn be used to in the future as a guide for system-level optimization, such as through cycle balancing. We demonstrated our method via a tool. Experimental results

show significant improvement over previously published results in terms of both the size of state space and run time.

Our proposed approach effectively views an asynchronous system as cyclical, compositional and recursive in structure, and periodic in its dynamics. This view has facilitated us to derive an efficient method to analyze system performance, and to define meaningful performance metrics for optimization. In contrast, existing work on the performance analysis of asynchronous systems often views the cyclic structure of these systems as an undesirable property and seek to analyze them as acyclic by unfolding their corresponding marked graph model.

We have offered a new, and in our view, potentially more suitable, way of looking at asynchronous systems, which we believe would lead to our ultimate goal of building optimal systems. In more detail, our compositional method for constructing the state space of the system under investigation based on their periodic property is analogous to the coupling of a system of oscillators. Oscillator models have been used for modeling various systems in the natural sciences and in engineering, for example, in robotics and in distributing clock signals in clocked systems, as well as in models of turbulence.

We believe that investigating this connection further can lead to interesting methods for engineering the design and synthesis of efficient asynchronous systems.

We see many avenues for further investigation. Research goals in the immediate future include extensions to analyze asynchronous systems with choice, the development of performance optimization algorithms for asynchronous systems driven by our analysis technique, and the application of our method to a broader class of concurrent systems, such as GALS and embedded systems.

In addition, a number of social and biological systems exhibit continuous-time behavior, with highly-concurrent operation and loosely-coupled synchronization. These include stochastic communication systems, and various swarming and flocking behaviors (e.g. fish, birds) in nature, which appear as self-regulating. Fundamentally, the notion of concurrent models, composed of simple cyclical operating units that are inter-coupled to form large-scale organizational behavior, which converges to steady-state periodicity, is a rich domain, and which suggests a great potential of discovery in applying computing models to real-world phenomena. It also suggests new paradigms to to understand the "self-clocking", and stochastic timing, of such systems, where the organizational movement comes from the consensus of individual distributed simpler behaviors, and no central control exists. *(Happy 60th Birthday, Alex!)*

References

1. Nowick, S.M., Singh, M.: Asynchronous design – part 1: overview and recent advances. IEEE Design & Test 22:3, 5–18 (2015)
2. Nowick, S.M., Singh, M.: Asynchronous design – part 2: systems and methodologies. IEEE Design & Test 22:3, 19–28 (2015)
3. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. IEEE Trans. Computers 47:8, 859–882 (1998)

4. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Trans. Information and Systems E80-D:3, 315–325 (1997)
5. Pang, P.B.K., Greenstreet, M.: Self-timed meshes are faster than synchronous. Proc. IEEE Async. Symp., 30–39 (1997)
6. Commoner, F., Holt, A.W., Pnueli, A.: Marked directed graphs. J. Comput. System Sciences 78, 511–523 (1971)
7. McGee, P.B., Nowick, S.M., Coffman Jr., E.G.: Efficient performance analysis of asynchronous systems based on periodicity. Proc. CODES/ISSS Symp., 225–230 (2005)
8. McGee, P.B.: On the timing behavior of concurrent digital systems: analysis, tools and applications. PhD Dissertation (Columbia University, CS Dept.) (2009)
9. Hulgaard, H., Burns, S.M., Amon, T., Borriello, G: An algorithm for exact bounds on the time separation of events in concurrent systems. IEEE Trans. Computers 44:11, 1306–1317 (1995)
10. Xie, A., Beerel, P.A.: Symbolic techniques for performance analysis of asynchronous systems based on average time separation of events. Proc. IEEE Async. Symp., 64–75 (1997)
11. Xie, A., Kim, S., Beerel, P.A.: Bounding average time separations of events in stochastic timed Petri nets with choice. Proc. IEEE Async. Symp., 94–107 (1999)
12. McGee, P.B., Nowick, S.M.: Discrete Event System (DES) Analyzer [part of public domain CaSCADE asynchronous tool package]. (for download access, see `http://www.cs.columbia.edu/~nowick/asynctools`)
13. Sutherland, I.E.: Micropipelines. Comm. ACM 32:6, 720–738 (1989)
14. Benes, R., Nowick, S.M., Wolfe, A.: A fast asynchronous Huffman decoder for compressed-code embedded processors. Proc. IEEE Async. Symp., 43–56 (1998)
15. Yun, K.Y., Beerel, P.A., Vakilotojar, V., Dooply, A.E., Arceo, J.: The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. IEEE Trans. VLSI Systems 6:4, 643–655 (1998)
16. Tugsinavisut, S., Hong, Y., Kim, D., Kim, K., Beerel, P.A.: Efficient asynchronous bundled-data pipelines for DCT matrix-vector multiplication. IEEE Trans. VLSI Systems 13:4, 448–461 (2005)