Efficient Timing Analysis of a Class of Petri Nets*

Henrik Hulgaard and Steven M. Burns

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
E-mail: {henrik,burns}@cs.washington.edu

Abstract. We describe an algebraic technique for performing timing analysis on a restricted class of Petri nets with interval time delays specified on the places of the net. The timing analysis we perform determines the extreme separation in time between specified occurrences of pairs of transitions for all possible timed executions of the system. We present the details of the timing analysis algorithm and demonstrate polynomial running time on a non-trivial parameterized example. Petri nets with 3000 nodes and 10^{16} reachable states have been analyzed using these techniques.

1 Introduction

The majority of research involving the formal analysis of temporal issues in concurrent systems has focused on powerful models of concurrency and these techniques are therefore often prohibitively computationally expensive. This paper takes the approach of using a less expressive model of a concurrent system in favor of a more efficient analysis. Our model of a concurrent system is based on safe Petri nets annotated with timing information. We will place restrictions on choice in order to make the set of possible executions independent of timing. This allows for a nice separation between the timing analysis and the enumeration of the possible execution paths.

This paper presents the details of an algorithm that determines the extreme case separation in time between two given transitions in a Petri net specification over all timed executions. By answering this separation problem, one can solve a number of problems in timing analysis, performance analysis and timing verification [10, 11]. For example, the best and worst case cycle period of a system is the minimum and maximum time, respectively, from a transition to the next occurrence of the same transition.

Related work in timing analysis and verification of concurrent systems comes from a variety of different research communities including: real-time systems,

* This work is supported by an NSF YI Award (MIP-9257987) and by the DARPA/CSTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041).

VLSI CAD, and operations research. Timed automata [1] is one of the more powerful models for which automated verification methods exists. A timed automaton has a number of *clocks* (timers) whose values can be used in guards of the transitions of the automaton. Such models have been extensively studied and several algorithms exist for determining timing properties for timed automata [7, 8]. As in the untimed case, timed automata suffer from the state explosion problem when constructing the cross product of component specifications. Furthermore, the verification time is proportional to the product of the maximum value of the clocks and also proportional to the number of permutations of the clocks.

To improve the run-time complexity, Burch [6] extends trace theory with discrete time but still uses automata-based methods for verification. This approach also suffers from exponential runtime in the size of the delay values but avoids the factorial associated with the permutations of the clocks. *Orbits* [16] uses convex regions to represent sets of timed states and thus avoids the explicit enumeration of each individual discrete timed state. *Orbits* is based on a Petri net model augmented with timing information. Other approaches that fall in this category include Timed Petri net [15] and Time Petri nets [3]. In Timed Petri nets a fixed delay is associated with each transitions while Time Petri nets use a more general model with delay ranges associated with the transitions.

2 Specifications

We use safe Petri nets to model concurrent systems. A *net* N is a tuple (S, T, F), where S and T are finite, disjoint, nonempty sets of respectively *places* and *transitions*, and $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation*. A *Petri net* Σ is a pair (N, M_0), where N is a net and $M_0 : S \longrightarrow \mathbf{N}$ is the *initial marking*. A Petri net is safe if for all reachable markings M and all places s, $M(s) \leq 1$. See [14] for further details on the Petri net model.

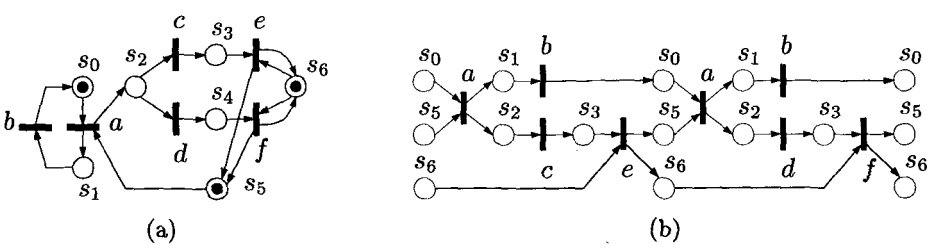


Fig. 1. (a) A simple Petri net Σ. The place s_2 is free choice and s_6 is unique choice. (b) A process π for Σ. The places and transitions in the process have been labeled (using the *lab* function) with the names of their corresponding places and transitions in Σ.

For an element $x \in S \cup T$, the *preset* and *postset* of x are defined as $\bullet x = \{y \in (S \cup T) : (y, x) \in F\}$ and $x\bullet = \{y \in (S \cup T) : (x, y) \in F\}$, respectively. We restrict the choice in the Petri net in order to simplify timing analysis. For each choice place $s \in S$, i.e., $|s\bullet| > 1$, we restrict s to be either *extended free choice* or *unique choice*. A place s is extended free choice if $\forall s' \in S : s\bullet \cap s'\bullet = \emptyset \vee s\bullet = s'\bullet$. The place s is unique choice if at most one of the successor transitions ever becomes enabled. Let $\#_t(s, M)$ denote the number of transitions in $s\bullet$ that are enabled at the marking M. A place s is unique choice if $\forall M \in [M_0\rangle : \#_t(s, M) \leq 1$. Note that unlike the extended free choice requirement, this is not a structural property of the Petri net. These restrictions still allow the analysis of a large and interesting class of Petri net specifications, including all live safe free choice nets. Fig. 1(a) shows a Petri net satisfying these restrictions.

3 Execution Semantics and Problem Definition

3.1 Processes

A *process* $\pi = (N, lab)$ for the Petri net Σ is a net N and a labeling $lab : S_\pi \cup T_\pi \longrightarrow S_\Sigma \cup T_\Sigma$. (We subscript S, T, and F to distinguish between the nets of Σ and π.) N and lab must satisfy appropriate properties such that π can be interpreted as an execution of Σ [4, 18]. Intuitively, the process $\pi = (N, lab)$ is an unfolding of Σ where all choice has been resolved, i.e., N is acyclic and choice free. Fig. 1(b) shows a process for the Petri net in Fig. 1(a). The only real choice is at the place s_2 where there is a non-deterministic selection of either transition c or d. The process represents the execution where the first time transition c fires and the next time transition d fires. We denote all (untimed) executions of a Petri net by the set

$$\Pi(\Sigma) = \{\pi \mid \pi \text{ is a process for } \Sigma\}.$$

For a live net, this is an infinite set. However, a safe Petri net has only a finite number of reachable markings. Processes have the property that any *cut* of places corresponds to a reachable marking of Σ [4, Lemma 2.7]. Therefore, sufficiently long processes will contain repeated segments of processes. We represent the infinite set of processes $\Pi(\Sigma)$ by a finite graph we call the *process automaton*. The vertices of the process automaton correspond to markings of Σ and the edges are annotated with segments of processes. We let v_0 denote the vertex corresponding to the initial marking M_0. Consider a path p in a process automaton from vertex u to v, denoted $u \overset{p}{\rightsquigarrow} v$. Then $\pi(p)$ is the process obtained by concatenating the process segments annotated on the edges of p. The process automaton has the property that

$$\Pi(\Sigma) = \bigcup \{pref(\pi(p)) \mid v_0 \overset{p}{\rightsquigarrow} v \text{ is a path in the process automaton}\},$$

where $pref(\pi)$ is the set of prefixes (defined on partial orders [18]) of a process π. We can construct the process automaton without first constructing the reachability graph [5, 9]. If there is no concurrency in the net, the size of the process

automaton is equal to the size of the reachability graph. However, if there is a high degree of concurrency, the process automaton will be considerably smaller. Fig. 2 shows the process automaton and the associated processes for the Petri net in Fig. 1(a). The process in Fig. 1(b) is constructed from $\pi_1\pi_3\pi_1\pi_2$.

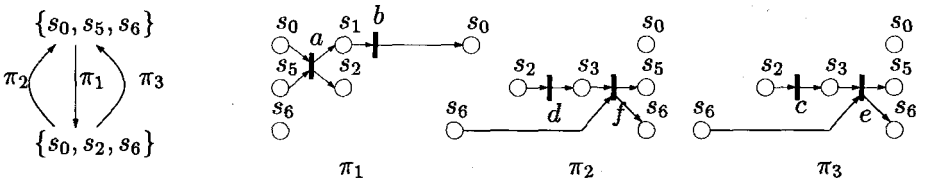


Fig. 2. To the left, the process automaton for the Petri net in Fig. 1(a). The three process segments annotated on the edges are shown to the right (labeled with elements from $S_\Sigma \cup T_\Sigma$).

3.2 Timed Execution

To incorporate timing into the Petri net model, we associate delay bounds with each place in the net. The lower delay bound, $d(s)$, and the upper delay bound, $D(s)$, are real numbers ($D(s)$ is allowed to be ∞) satisfying $0 \leq d(s) \leq D(s)$. These delay bounds restrict the possible executions of the Petri net. During a timed execution of the net, when a token is added to a place s, the earliest it becomes available for a transition in $s\bullet$ is $d(s)$ time units later and the latest is $D(s)$ units later. A transition t must fire when there are available tokens at all places in $\bullet t$ unless the firing of the transition is disabled by firing another transition. The firing of t itself is instantaneous.

More formally, a *timing assignment* for process π, τ, is a function that maps transitions in a process to time values, $\tau : T_\pi \longrightarrow \mathbf{R}^{\neq}$, where $\mathbf{R}^{\neq}$ is the set of non-negative real numbers. Let Σ be a Petri net and let π be a process of Σ. We consider a cut $c \subseteq S_\pi$ of π and let $T_{\text{enabled}} \subseteq T_\Sigma$ be the set of transitions enabled at the corresponding marking, M_c. For a timing assignment, τ, and a transition $t \in T_{\text{enabled}}$, the earliest and latest global firing time of t is given by

$$\begin{aligned} \mathit{earliest}(t) &= \max\{\mathit{starttime}(b) + d(\mathit{lab}(b)) \mid b \in c \cap \mathit{lab}^{-1}(\bullet t)\} \quad \text{and} \\ \mathit{latest}(t) &= \max\{\mathit{starttime}(b) + D(\mathit{lab}(b)) \mid b \in c \cap \mathit{lab}^{-1}(\bullet t)\}, \end{aligned}$$

where $\mathit{lab}^{-1}(s)$ denotes the set of elements of S_π which are mapped to s by lab. Note that $c \cap \mathit{lab}^{-1}(\bullet t)$ is non-empty because t is enabled at marking M_c. The function $\mathit{starttime}$ takes a place $b \in S_\pi$ and returns the time when a token entered the place $\mathit{lab}(b)$, i.e., $\tau(e)$ if $(e, b) \in F_\pi$. If there is no such transition e, we set $\mathit{starttime}(b)$ to 0. The timing assignment τ is *consistent at cut* c if for all $e \in c\bullet$:

1. $earliest(lab(e)) \leq \tau(e) \leq latest(lab(e))$, and

2. $\forall t \in T_{\text{enabled}} \setminus lab(c\bullet) : \tau(e) \leq latest(t)$.

Condition 1 states that the delay bounds must be respected. Condition 2 states that for those transitions t that are enabled but do not have a corresponding transition included in π, t must not have been required to fire previously because of the restrictions imposed by the timing bounds.

A timing assignment τ of a process is consistent if it is consistent at all cuts c of the process. Let

$$\Pi_{\text{timed}}(\Sigma) = \bigcup \{pref(\pi) \mid \pi \in \Pi(\Sigma) \text{ and there exists a consistent } \tau \text{ for } \pi\}.$$

The restrictions on the Petri net in Section 2 were crafted so that the set of untimed and timed processes are equivalent. This allows us to use the process automaton to enumerate the possible processes without referring to timing information, and then perform timing analysis on each process individually.

Theorem 1. *Let Σ be a safe Petri net where the choice is either extended free choice or unique choice. Then $\Pi_{\text{timed}}(\Sigma) = \Pi(\Sigma)$.*

Proof. (Sketch) By the definitions and the fact $\Pi(\Sigma)$ is prefix-closed, we have $\Pi_{\text{timed}}(\Sigma) \subseteq \Pi(\Sigma)$. We need to show that $\Pi(\Sigma) \subseteq \Pi_{\text{timed}}(\Sigma)$, i.e., for all $\pi \in \Pi(\Sigma)$ we can find a $\pi' \in \Pi(\Sigma)$ and a consistent timing assignment for π' such that $\pi \in pref(\pi')$. Condition 1 can always be satisfied by choosing τ appropriately for a particular π'. Some processes do not have a timing assignment because they do not contain enabled transitions that are required to have fired by condition 2. We must show that any π to be extended with these transitions. This can always be done except in conflict cases that have been explicitly eliminated by the choice restrictions.

3.3 Problem Formulation

Given two transitions from a restricted Petri net Σ, $t_{\text{from}}, t_{\text{to}} \in T_\Sigma$, we wish to determine the extreme-case separation in time between related firings of t_{from} and t_{to}. We let $\widehat{\Pi}$ be a set of triples $\widehat{\pi} = \langle \pi, t_{\text{src}}, t_{\text{dst}} \rangle$, where $\pi \in \Pi(\Sigma)$ and $t_{\text{src}}, t_{\text{dst}}$ are transitions in the process π with $lab(t_{\text{src}}) = t_{\text{from}}$, $lab(t_{\text{dst}}) = t_{\text{to}}$. The set $\widehat{\Pi}$ is used to describe all the possible processes where the distinguished transitions t_{src} and t_{dst} have the appropriate relationship. This relationship must be established in order for the timing analysis to yield interesting information [10].

Consider finding the maximum time between consecutive firings of transition a in Fig. 1(a), i.e., $t_{\text{from}} = t_{\text{to}} = a$. All the elements of $\widehat{\Pi}$ must have the property that no other transition t between t_{src} and t_{dst} has label a. One of the elements in $\widehat{\Pi}$ is the process in Fig. 1(b) with t_{src} and t_{dst} being the left-most and right-most transitions labeled with a, respectively.

The timing analysis we perform is: for all $\widehat{\pi} \in \widehat{\Pi}$, and for all consistent timing assignments τ for π, determine the largest δ and smallest Δ such that

$$\delta \leq \tau(t_{\text{dst}}) - \tau(t_{\text{src}}) \leq \Delta .$$

In the sequel, we will only discuss the *maximum* separation analysis, i.e., find Δ, because the minimum separation δ can be found from a maximum separation analysis of $\tau(t_{\text{src}}) - \tau(t_{\text{dst}}) \leq -\delta$.

4 Timing Analysis

Let $\Delta(\widehat{\pi})$ be the maximum separation between t_{src} and t_{dst} for some particular execution $\widehat{\pi}$:

$$\Delta(\widehat{\pi}) = \max\{\tau(t_{\text{dst}}) - \tau(t_{\text{src}}) \mid \tau \text{ is a consistent timing assignment for } \pi \} .$$

The maximum separation over all executions is then given by

$$\Delta = \max\{\Delta(\widehat{\pi}) \mid \widehat{\pi} \in \widehat{\Pi}\} . \tag{1}$$

This section shows how the elements of $\widehat{\Pi}$ are constructed to obtain Δ, and Section 5 describes the algorithm for computing $\Delta(\widehat{\pi})$.

4.1 Constructing the Processes of $\widehat{\Pi}$

The process automaton represents all possible executions starting at the initial marking (v_0 in the process automaton). However, whatever follows the t_{src} and t_{dst} in a process cannot influence the maximum separation between these two transitions. We can therefore ignore any portion of a process following t_{src} and t_{dst}. All processes in $\widehat{\Pi}$ will therefore end with some terminal process segment that includes the two transitions t_{src} and t_{dst}. We can construct a finite set of terminal process segments such that all processes in $\widehat{\Pi}$ can be constructed from some path in the process automaton followed by one of the terminal process segments. Fig. 3 shows the two terminal process segments for the a-to-a separation analysis in our example.

4.2 The CTSE Algorithm

An algorithm for computing $\Delta(\widehat{\pi})$ can be phrased in algebraic terms. For each *segment* of a process, there is a corresponding element in the algebra. We use $[\pi]$ to denote this element for the process segment π. The algebra allows us to reuse analysis of shorter processes when computing $\Delta(\widehat{\pi})$ because the operators of the algebra are associative (the details are shown in the next section). There are two operations in the algebra: "choice", $|$, and "concatenation", $\odot$.

Our approach to analyzing the infinite set $\widehat{\Pi}$ is to enumerate the processes $\widehat{\pi}$ of increasing length by unfolding the process automaton using a breadth-first traversal. We traverse the automaton *backwards*, starting with the terminal

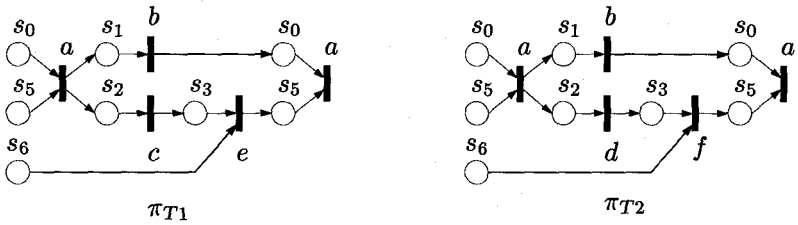
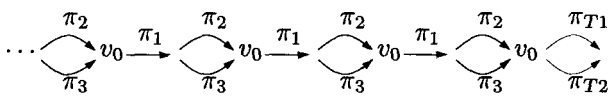


Fig. 3. Two terminal processes (labeled using *lab*) for the separation analysis from a to the next a transition in the Petri net in Fig. 1(a).

segments. At each node v in the process automaton, we store an element of the algebra. Let $[v]_k$ denote the algebraic element stored at node v in the process automaton after the k^{th} iteration. Initially, $[v]_0 = [\pi_T]$ for each terminal segment exiting node v. If there is more than one terminal segment exiting v, the algebraic elements are combined using the choice-operator.

When traversing the process automaton backwards, the elements of the algebra are composed (using $\odot$) for two paths in series, and combined (using $|$) for two paths in parallel. The choice-operator combines backward paths when they reach the same marking in the process automaton. This is illustrated below by showing a backward traversal with reconvergence corresponding to the process automaton in Fig. 2 and the two terminal processes in Fig. 3:



For this example, $[v_0]_0 = [\pi_{T1}] \mid [\pi_{T2}]$ and for all $k \geq 1 : [v_0]_k = [\pi_1] \odot ([\pi_2] \mid [\pi_3]) \odot [v_0]_{k-1}$. Whenever the node v_0 is reached in the k^{th} unfolding, $[v_0]_k$ represents the maximum separation for all executions represented by that unfolding, denoted Δ_k. This value is maximized with the values for the previous unfoldings, $\Delta_{\leq k} = \max\{\Delta_i \mid 0 \leq i \leq k\}$. By definition (1), $\Delta_{\leq k}$ is a lower bound on Δ and $\Delta = \lim_{k \to \infty} \Delta_{\leq k}$.

For a given node v in the process automaton, we can compute an upper bound on all further unfoldings; this bound is denoted $[v]_{>k}$. Let c be a vertex cut of the process automaton. An upper bound on Δ for the k^{th} unfolding is $\Delta_{>k} = \max\{[v]_{>k} \mid v \in c\}$. When $\Delta_{>k}$ is less than or equal to $\Delta_{\leq k}$ for some k we can stop further unfolding and report the exact maximum separation $\Delta = \Delta_{\leq k}$. It is possible that the upper and lower bounds do not converge in which case the bounds may still provide useful information as Δ is in the range $[\Delta_{\leq k}, \Delta_{>k}]$. The main loop of the CTSE algorithm is shown in Figure 4.

The run-time of the algorithm depends on the size of the representation of the algebraic elements. The size of these elements may be as large as the number of paths between the two nodes related by the element, i.e., exponential in the number of iterations, k. In practice, pruning drastically reduces the element size.

```
ALGORITHM: CTSE(G)
    For each v ∈ G, place [π_T(v)] at v;
    Δ_{≤k} ← −∞;
    do {
      UNFOLD_ONCE(G);
      Δ_{≤k} ← max(Δ_{≤k}, [v_0]_k);
      Δ_{>k} ← max{[v]_{>k} | v ∈ cut of G};
    } until Δ_{≤k} ≥ Δ_{>k};
    return Δ_{≤k};
```

Fig. 4. The CTSE algorithm computing Δ given a process automaton G.

5 Computing $\Delta(\widehat{\pi})$

This section describes the algebra used in the CTSE algorithm. This algebra is used to reformulate an algorithm by McMillan and Dill [13] for determining the maximum separation of two events in an acyclic graph.

5.1 Algebras

Before presenting the algorithm for computing $\Delta(\widehat{\pi})$ we introduce two algebras. The first is the $(\min, +)$-algebra $(\mathbf{R} \cup \{\infty\}, \oplus', \otimes', \infty, 0)$ where

$$x \oplus' y = \min(x, y)\,, \qquad x \otimes' y = x + y\,.$$

The elements ∞ and 0 are the identity elements for $\oplus'$ and $\otimes'$, respectively.

The second algebra is denoted by $(\mathcal{F}, \oplus, \otimes, \overline{0}, \overline{1})$. Each element in $\mathcal{F}$ is a function represented by a set of pairs. The singleton set, $\{\langle l, \mathbf{u}\rangle\}$, where $\mathbf{u}$ is a row-vector of length n, represents the function:

$$f(x, \mathbf{m}) = \min(l + x, \mathbf{u} \otimes' \mathbf{m})\,,$$

where $\mathbf{m}$ is a column-vector of length n and $\otimes'$ denotes the inner product in the $(\min, +)$-algebra. In general, the set $\{\langle l_1, \mathbf{u}_1\rangle, \langle l_2, \mathbf{u}_2\rangle, \ldots, \langle l_n, \mathbf{u}_n\rangle\}$ represents the function

$$f(x, \mathbf{m}) = \max\left\{\min(l_i + x, \mathbf{u}_i \otimes' \mathbf{m}) \;\middle|\; 1 \le i \le n\right\}. \tag{2}$$

We associate two binary operators with functions: function maximization, $f \oplus g$, and function composition, $f \otimes g$. It follows from (2) that function maximization can be defined as set union: $f \oplus g = f \cup g$. Function composition, $f = g \otimes h$,

is defined as $f(x, \mathbf{m}) = h(g(x, \mathbf{m}), \mathbf{m})$. Notice that we use left-to-right function composition. For $g = \{\langle l_1, \mathbf{u}_1 \rangle\}$ and $h = \{\langle l_2, \mathbf{u}_2 \rangle\}$ we have

$$(g \otimes h)(x, \mathbf{m}) = h(g(x, \mathbf{m}), \mathbf{m}) \quad \text{and} \quad g \otimes h = \{\langle l_1 + l_2, (\mathbf{u}_1 + l_2) \oplus' \mathbf{u}_2 \rangle\}.$$

Function composition, $\otimes$, distributes over function maximization, $\oplus$. The elements $\overline{0}$ and $\overline{1}$ are the identity elements for function maximization and composition, respectively.

Let $p_i = \langle l_i, \mathbf{u}_i \rangle$ and $p_j = \langle l_j, \mathbf{u}_j \rangle$ be two pairs in the representation of a function. We can remove p_j if $l_i \geq l_j$ and $\mathbf{u}_i \geq \mathbf{u}_j$ (componentwise), since then for all x and $\mathbf{m}$, $\min(x + l_i, \mathbf{u}_i \otimes' \mathbf{m}) \geq \min(x + l_j, \mathbf{u}_j \otimes' \mathbf{m})$. Proper application of this observation that can greatly simplify the representation of a function.

5.2 The Acyclic Time Separation of Events Algorithm

We can now present the algebraic formulation of McMillan and Dill's algorithm for computing $\Delta(\widehat{\pi})$. For each place and transition in π we compute a pair $[f, m]$ where $f \in \mathcal{F}$ and $m \in \mathbf{R}$. Each $[f, m]$ pair is computed in backward topological order of $\widehat{\pi}$. For each place s, the pair $[f', m']$ at s is determined by:

$$[f', m'] = \begin{cases} [\overline{0}, \infty] & \text{if } s\bullet = \emptyset \\ [\{\langle D(s), (m) \rangle\} \otimes f, -d(s) \otimes' m] & \\ \qquad \text{where } [f, m] \text{ is the pair stored at } t \in s\bullet. & \end{cases}$$

For each transition t, we compute the pair $[f', m']$ as:

$$f' = \begin{cases} \overline{1} & \text{if } t = t_{\text{dst}} \\ \oplus\{f \text{ at place } s \mid s \in t\bullet\} & \text{otherwise} \end{cases}$$

$$m' = \begin{cases} 0 & \text{if } t = t_{\text{src}} \\ \oplus'\{m \text{ at place } s \mid s \in t\bullet\} & \text{otherwise} \end{cases}$$

Informally, this algorithm works as follows: To maximize the value of $\tau(t_{\text{dst}}) - \tau(t_{\text{src}})$ we need to find a timing assignment that maximizes $\tau(t_{\text{dst}})$ and minimizes $\tau(t_{\text{src}})$. The first element of $[f', m']$ represents the longest path (using $D(s)$) from a transition to t_{dst} and the second element represents the shortest path (using $-d(s)$) to t_{src}. The algebra for the f'-part is complicated by the fact that the delay for a given place can not be assigned both $d(s)$ and $D(s)$. The f'-part must represent the longest path respecting the delays assigned by the shortest path computation. For details see [12]. To find the maximum separation represented by a $[f, m]$ pair, we evaluate f at m and 0, computing the sum of the longest and shortest paths. To compute $\Delta(\widehat{\pi})$, we maximize over all $[f, m]$ pairs at the initial marking:

$$\Delta(\widehat{\pi}) = \max\{f(m, 0) \mid [f, m] \text{ is the pair at } s \in \bullet\pi\},$$

where $\bullet\pi$ denotes the set $\{s \in S_\pi \mid \bullet s = \emptyset\}$, and, similarly, $\pi\bullet$ denotes the set $\{s \in S_\pi \mid s\bullet = \emptyset\}$.

5.3 Decomposition

The algebraic formulation allows for a decomposition of the above computation using matrices. Consider a process segment π having $|\bullet\pi| = m$ and $|\pi\bullet| = n$. We represent the computation of the algorithm on π by two $n \times m$ matrices, $\mathbf{F}$ and $\mathbf{M}$. Given a vector of m-values at $\pi\bullet$, $\mathbf{m}$, we can find the vector m-values at $\bullet\pi$, $\mathbf{m}'$, from the $(\min, +)$-matrix multiplication $\mathbf{m}' = \mathbf{M} \otimes' \mathbf{m}$. The functions (the f-part), however, depend on the m-values at the places in $\pi\bullet$. This dependency is encoded by a vector $\mathbf{u}$ of length n. The vector product $\mathbf{u} \otimes' \mathbf{m}$ computes the shortest path to the internal node where $\mathbf{u}$ is stored. For example, in the process segment π_1 of Fig. 2 with all delay ranges set to $[1, 2]$, we get the two matrices

$$\mathbf{F} = \begin{pmatrix} \{\langle 4, (0\,0\,\infty)\rangle\} & \{\langle 2, (0\,0\,\infty)\rangle\} & \overline{0} \\ \{\langle 4, (0\,0\,\infty)\rangle\} & \{\langle 2, (0\,0\,\infty)\rangle\} & \overline{0} \\ \overline{0} & \overline{0} & \overline{1} \end{pmatrix} \quad \text{and} \quad \mathbf{M} = \begin{pmatrix} -2 & -1 & \infty \\ -2 & -1 & \infty \\ \infty & \infty & 0 \end{pmatrix}.$$

Given a process segment π, we denote the corresponding function and m-value matrices by $\mathbf{F}(\pi)$ and $\mathbf{M}(\pi)$. The algebraic element $[\pi]$ is then defined as the singleton set $\{[\mathbf{F}(\pi), \mathbf{M}(\pi)]\}$. We can now define the two operators $\odot$ and $|$. The choice operator is defined as set union:

$$[\pi_1] \mid [\pi_2] = [\pi_1] \cup [\pi_2].$$

The composition operator is more complex. When composing two segments π_1 and π_2, the functions in π_1 need to refer to the m-values in $\pi_2\bullet$ rather than those at $\pi_1\bullet$. We *shift* the functions in $\mathbf{F}(\pi_1)$ to make them refer to m-values in $\pi_2\bullet$ by multiplying the $\mathbf{u}$-vectors in $\mathbf{F}(\pi_1)$ with $\mathbf{M}(\pi_2)$. For a singleton function $\{\langle l, \mathbf{u}\rangle\}$, we obtain the function $\{\langle l, \mathbf{u} \otimes' \mathbf{M}(\pi_2)\rangle\}$. Non-singleton functions are shifted by shifting each pair, and a matrix of functions is shifted elementwise. We use the notation $\mathbf{F} \ll \mathbf{M}$ to denote a shift of matrix $\mathbf{F}$ by matrix $\mathbf{M}$. For singleton sets the composition operator is defined as:

$$\{[\mathbf{F}(\pi_1), \mathbf{M}(\pi_1)]\} \odot \{[\mathbf{F}(\pi_2), \mathbf{M}(\pi_2)]\} = \{[(\mathbf{F}(\pi_1) \ll \mathbf{M}(\pi_2)) \otimes \mathbf{F}(\pi_2), \mathbf{M}(\pi_1) \otimes' \mathbf{M}(\pi_2)]\}.$$

Non-singleton sets are multiplied out by applying the distributive law.

5.4 Pruning

Consider the element $\{[\mathbf{F}_1, \mathbf{M}_1], [\mathbf{F}_2, \mathbf{M}_2]\}$. We can removed $[\mathbf{F}_2, \mathbf{M}_2]$ from the set if we can show that for any fixed pairs composed to the left and right such that the result is a scalar, this scalar is no greater than the same composition with the $[\mathbf{F}_1, \mathbf{M}_1]$ pair. We have developed a sufficient condition for when $[\mathbf{F}_1, \mathbf{M}_1]$ prunes $[\mathbf{F}_2, \mathbf{M}_2]$. This condition is used to eliminate entire execution paths from further analysis, and is central to obtaining an efficient algorithm. More sophisticated conditions, that use more information about the particular computation, are possible and may further increase the efficiency of the algorithm.

5.5 Upper Bound Computation

We now consider how to determine an upper bound $[v]_{>k}$ for node, v, in the process automaton. To determine a non-trivial upper bound, all further backward paths from v to v_0 have to be considered, i.e., we need to bound the infinite set of processes constructed from backward paths p:

$$\left\{\pi(p) \mid v_0 \overset{p}{\rightsquigarrow} v\right\}. \tag{3}$$

For any simple path p we just compute $[\pi(p)]$. If p is not simple, we write p as $p = p_1 p_2^n p_3$, where p_3 is a simple path, p_2 is a simple cycle, and p_1 is finite, but may contain cycles. We introduce an *upper bound operator*, ∇, with the property that

$$[\pi_1] \odot [\pi_2]^n \odot [\pi_3] \leq [\pi_1] \odot [\pi_3] \mid (\nabla[\pi_2]) \odot [\pi_3],$$

where $\pi_i = \pi(p_i)$ for $i = 1, 2, 3$ and $n \geq 1$. By applying this lemma, the cycle p_2 is removed from path p. The ∇ operator is recursively applied to the path $p_1 p_3$ until this is a simple path. Hence, we can bound the infinite set in (3) by a finite set of paths constructed from a simple cycle followed by a simple path.

Applying ∇ to the pair $[\mathbf{F}, \mathbf{M}]$ sets all elements of $\mathbf{M}$ to infinity and applies the function $z = \{\langle \infty, \overline{\infty}_k \rangle\}$ to all elements in the dense part of $\mathbf{F}$ (corresponding to the part of the process segment without isolated places.) $\overline{\infty}_k$ is a row-vector of length k whose elements are all ∞. This element has the property that for all functions f, $z \geq f$.

The upper bound is determined individually for each pair in the set for node v. If the upper bound for a given $[\mathbf{F}, \mathbf{M}]$ pair is less than or equal to the present global lower bound, $\Delta_{\leq k}$, that pair can be removed from the set for node v, further pruning the backward execution paths that must be considered.

The order in which $[\mathbf{F}, \mathbf{M}]$ pairs are multiplied greatly affects the run-time of the algorithm. For example, consider precomputing for each node in the process automaton the algebraic expression for the upper bound, i.e., for each node, compute the algebraic element for the set of simple paths followed by simple cycles (going backwards). Because we don't know what is to be composed with these elements, few pairs can be pruned from the representation. Because of this, it may be more efficient to multiply the pairs out in each iteration, even though this doesn't allow the reuse of work from previous iterations. Our experience has been that upper bound expressions become very large when precomputed and we are better off recomputing them at each iteration because effective pruning takes place. We only precompute the ∇ of the simple cycles. This observation was key to achieving polynomial run-time for the example described in the following section.

6 Benchmark Example: The Eager Stack

Replicating a single process in a linear array provides an efficient hardware implementation of a last-in, first-out memory which we refer to as an *eager stack* [10].

The eager stack contains an interesting mixture of choice and concurrency and represents an excellent parameterizable example for benchmarking our implementation of the algorithm.

A stack capable of storing n elements is constructed from n identical processes arranged in a linear array. There are numerous interesting time separation analysis we can perform on the eager stacks. Here we show execution times for two maximum separation analyses. One is the maximum separation between consecutive push operations; the other is the maximum separation between consecutive push or pop operations, corresponding to the maximum response time of the stack. Table 1 and Fig. 5 shows execution times of the CTSE algorithm for these two separation analyses on eager stacks of various sizes where all delay ranges are set to $[1, 2]$.

Table 1. Execution times of the CTSE algorithm on eager stacks of various sizes, n. All delay ranges are $[1, 2]$. The size of the specification, i.e., number of places, number of transitions, and the size of the flow relation, is given by $|S_\Sigma|$, $|T_\Sigma|$, and $|F_\Sigma|$, respectively. The number of nodes in the reachability graph is shown in the $|R.G.|$ column. (Note that the reachability graph is not constructed when performing the timing analysis and is only reported to give an idea of the complexity of the nets.) The separation analysis denoted by Δ_1 is the maximum separation between consecutive push operations and Δ_2 is the maximum separation between consecutive push or pop operations. The CPU times were obtained on a Sparc 10 with 256 MB of memory.

Size (n)	$\lvert S_\Sigma\rvert$	$\lvert T_\Sigma\rvert$	$\lvert F_\Sigma\rvert$	$\lvert R.G.\rvert$	CPU Δ_1 (secs)	CPU Δ_2 (secs)
3	20	13	53	14	.4	.3
4	32	21	86	36	.7	.6
5	47	31	127	97	1.8	1.3
6	65	43	176	268	2.4	2.3
8	110	73	298	2124	6.8	6.6
10	167	111	452	$2 \cdot 10^4$	16	17
12	236	157	638	$14 \cdot 10^4$	34	31
16	410	273	1106	$9 \cdot 10^6$	116	111
20	632	421	1702	$6 \cdot 10^8$	303	296
24	902	601	2426	$4 \cdot 10^{10}$	678	719
28	1220	813	3278	$2 \cdot 10^{12}$	1404	1494
32	1586	1057	4258	$2 \cdot 10^{14}$	2723	2954
36	2000	1333	5366	$1 \cdot 10^{16}$	4510	4858

Orbits [16] is, to the authors knowledge, the most developed and efficient tool for answering temporal questions about Petri nets specifications. *Orbits* constructs the timed reachability graph, i.e., the states reachable given the timing information. It should be noted that *Orbits* is capable of analyzing a larger class of Petri net specifications than the one described here. Partial order techniques

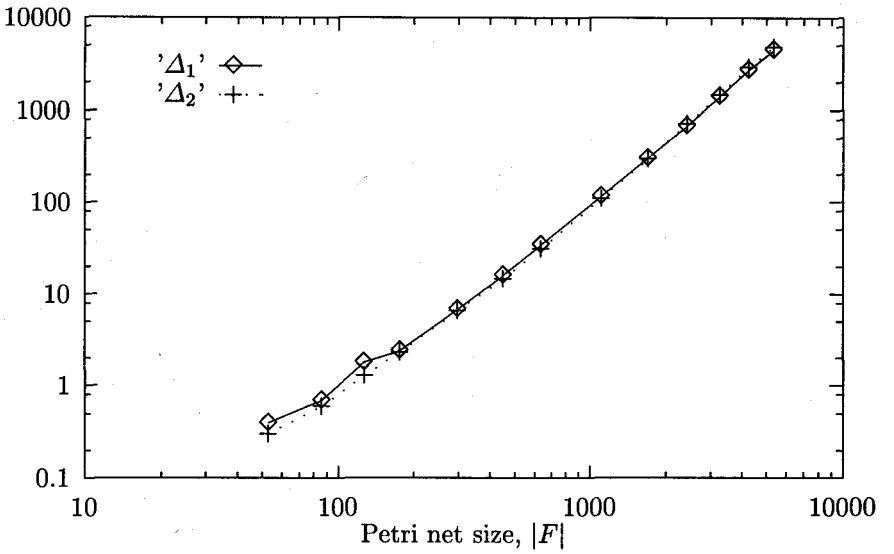


Fig. 5. Double logarithmic plot of CPU time for the two separation analyses as a function of the Petri net size, $|F|$.

are also used in *Orbits* to reduce the state space explosion [17]. However, the time to construct the timed reachability graph for the eager stack increases exponentially with the stack size n. For $n = 6$ the time is 234 CPU seconds on a Decstation 5000 with 256 MB, i.e., two orders of magnitude slower than the CTSE algorithm. For $n = 7$, *Orbits* ran out of memory.

7 Conclusion

We have described an algorithm for solving an important time separation problem on a class of Petri nets that contains both choice and concurrency. In practice, our algorithm is able to analyze nets of considerable size, demonstrated by an example whose Petri net specification consists of more than 3000 nodes and 10^{16} reachable states. While we report a polynomial run-time result for only a single parameterizable example, we expect similar results for other specifications exhibiting limited choice and abundant concurrency.

Acknowledgments

The authors thank the anonymous reviewers for their thoughtful and detailed comments.

References

1. R. Alur and D. L. Dill. The theory of timed automata. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rosenberg, editors, *Real-Time: Theory in Practice*, LNCS #600, pages 28–73. Springer-Verlag, 1991.
2. F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity.* John Wiley and Sons, 1992.
3. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.
4. E. Best and J. Desel. Partial order behavior and structure of Petri nets. *Formal Aspects of Computing*, 2:123–138, 1990.
5. E. Best and Raymond Devillers. Interleaving and partial orders in concurrency: A formal comparison. In M. Wirsing, editor, *Formal Description of Programming Concepts–III*, pages 299–323, 1986.
6. J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* Ph.D. thesis, Carnegie Mellon, August 1992.
7. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1:385–415, 1992.
8. D. L. Dill, editor. *Computer-Aided Verification '94.*
9. P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, pages 321–340.
10. H. Hulgaard and S. M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.
11. H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. Practical applications of an efficient time separation of events algorithm. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 146–151, November 1993.
12. H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. To appear in IEEE Transactions on Computers. Available as University of Washington CS&E Technical Report #94-02-02.
    (anonymous ftp: `cs.washington.edu:tr/1994/02/UW-CSE-94-02-02.PS.Z`)
13. K. L. McMillan and D. L. Dill. Algorithms for interface timing verification. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1992.
14. J. L. Peterson. *Petri Net Theory and The Modeling of Systems.* Prentice-Hall, 1981.
15. C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, SE-6(5):440–448, September 1980.
16. T. G. Rokicki. *Representing and Modeling Digital Circuits.* Ph.D. thesis, Stanford University, 1993.
17. T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In D. L. Dill, editor, *Computer-Aided Verification '94*, pages 468–480.
18. W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets.* LNCS #625. Springer-Verlag, 1992.