An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems[1]

Henrik Hulgaard, Steven M. Burns,
Tod Amon[2], and Gaetano Borriello

Department of Computer Science and Engineering, FR–35
University of Washington
Seattle, WA 98195

Technical Report 94-02-02
February 9, 1994

[1]Submitted to IEEE Transactions on Computers.
[2]Department of Computer Science, Southwest Texas State.

An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems[1]

Henrik Hulgaard, Steven M. Burns, Tod Amon*, and Gaetano Borriello

Department of Computer Science and Engineering, FR–35
University of Washington
Seattle, WA 98195
E-mail: {henrik,burns,gaetano}@cs.washington.edu

*Department of Computer Science
Southwest Texas State
San Marcos, TX 78666–4616
E-mail: ta02@academia.swt.edu

Submitted to IEEE Transactions on Computers

February 9, 1994

Abstract

Determining the time separation of events is a fundamental problem in the analysis, synthesis, and optimization of concurrent systems. Applications range from logic optimization of asynchronous digital circuits to evaluation of execution times of programs for real-time systems. We present an efficient algorithm to find exact (tight) bounds on the separation time of events in an arbitrary process graph without conditional behavior. This result is more general than the methods presented in several previously published papers as it handles cyclic graphs and yields the tightest possible bounds on event separations. The algorithm is based on a functional decomposition technique that permits the implicit evaluation of an infinitely unfolded process graph. Examples are presented that demonstrate the utility and efficiency of the solution. The algorithm will form a basis for exploration of timing-constrained synthesis techniques.

Index terms: Abstract algebra, asynchronous systems, concurrent systems, discrete event systems, time separation of events, timing verification.

1 Introduction

Computer systems are becoming increasingly more concurrent. This is true with regards to both their implementation and their specification. Implementations are becoming more concurrent because the concurrency provides a mechanism for improving response time by performing multiple computations at the same time. Specifications are often concurrent, because the concurrency provides a means by which the design's complexity can be broken up into smaller components that interact with one another. Unfortunately, concurrent systems are notoriously difficult to design. Managing the large number of interactions occurring concurrently necessitates the development of formal methods for specifying and analyzing the behavior of these systems.

For these reasons there is a growing interest in formal verification. Verification tools are used to formally reason about system correctness, and reasoning about complex system behavior often involves reasoning about time. Since design automation tools are working with higher levels of design abstraction, timing issues are becoming especially important. Many aspects of a system's behavior are not under the control of the designer. Systems must conform to the environment in which they will be placed. The environment may demand that particular timing relationships be respected. If synthesized designs need to be integrated into pre-existing systems, then synthesis tools must accommodate the environment and the timing relationships that need to be satisfied.

When a design is specified at a high level, important trade-offs are made during synthesis. There are many possible implementations; it may be that part of the design can be in hardware and part in software. One important criterion that can be used in making such decisions is an analysis of the resulting timing behavior of a system and whether it will meet the timing constraints provided by the designer. Verification tools are thus needed not only after synthesis (i.e., implementation verification) but also during synthesis, for they can provide guarantees that will enable synthesis algorithms to make good choices and discover possible optimizations. For example, timing information can be used to help determine an appropriate partitioning for a design, and is also of vital importance in scheduling.

Related work in timing verification comes from a variety of different research communities, and is quite extensive. One aspect which much of the research has in common is its focus on analyzing very general systems. This is true for approaches based on decision procedures (e.g., [16, 17]), model checking (e.g., [7, 18]), exhaustive simulation (e.g., [20]), language containment (e.g., [2, 26]), deductive proofs techniques (e.g., [27]) and theorem proving (e.g., [12]). For this reason, the timing analysis is usually at least of exponential time complexity and is often undecidable [15]. Because these approaches do not yield practical and efficient algorithms they are not particularly useful for synthesis purposes. In contrast, some synthesis and timing verification tools greatly restrict the classes of behavior which can be analyzed in favor of efficient verification algorithms.

Interface verification algorithms attempt to determine minimum and maximum separation times between system events given the propagation delays of the specified system. An event can be thought of as an execution point in the system (i.e., a completion or initiation of a computation or a synchronization). Depending on the level of abstraction in the specification, events may represent low-level signal transitions at a circuit interface or abstract behavioral control flow.

This paper presents an algorithm that determines tight upper and lower bounds on the separation in time of an arbitrary pair of events in a concurrent system. The algorithm achieves its efficiency by restricting the class of concurrent systems to those without conditional behavior. Fortunately, that still leaves a large and useful class of deterministic concurrent specifications to which our analysis applies. To illustrate the kinds of systems that can be analyzed, consider the

following concurrent system consisting of three processes that synchronize over two channels a and b, and perform some internal computation (delay ranges specified in brackets):

```
Process 1 ::                Process 2 ::                Process 3 ::

  repeat {                    repeat {                    repeat {
      Synchronize a;              Synchronize a;              Synchronize b;
      Compute [4, 10];            Compute [1, 2];             Compute [5, 20];
  }                               Synchronize b;          }
                                  Compute [1, 6];
                              }
```

There are many questions regarding the temporal behavior of this system that we might ask: "How slowly might the first process cycle?", or "How long might the second process be idle waiting on the third process?", or "How can we best speed-up the performance of our system?" and "Which delays impact performance the most?" To answer such questions we need to be able to determine how the inter-process synchronizations affect the temporal behavior of our concurrent system. For example, the first process could obviously have a cycle period of at least 10 time units (the upper bound on the computation time) but how much more delay might be incurred as a result of the synchronization with the second process?

Other approaches to the problem of efficiently finding bounds on the separation in time of two events have either been inexact or based on a more restrictive graph topology. Loose bounds that may not enable all possible optimizations were obtained by [25]. Both [24] and [30] handle only acyclic graphs. However, they provide a theoretical foundation upon which our solution is built. Both [8] and [6] use cyclic graph models but they deal only with fixed delays between events. Several verifiers have been developed for specific applications [3, 5, 21].

This paper is composed of six sections. We follow this introduction with a formalization of the timing analysis and in Section 3 we present an algorithm that solves the maximum separation problem for acyclic graphs. This algorithm is then extended in Section 4 to handle cyclic graphs by introducing a functional algebra that allows us to implicitly analyze an infinite graph. Section 5 presents four applications of the timing analysis and finally Section 6 summarizes the contributions.

2 Problem Formalization

2.1 The Process Graph

We represent a concurrent system as a directed graph, called the *process graph*. The process graph is a simple modification of the *event–rule system* developed in [6]. The model can also be viewed as an extension of [24] and [30], where we consider cyclic max-only or type-2 graphs, respectively. Process graphs are similar to timed event graphs [4], marked graphs or decision free Petri nets. Let $G' = \langle E', R' \rangle$ denote a process graph composed of

- a finite set of events, E', the vertices of the graph.
- a finite set of rule templates, R', the edges of the graph.

Each edge is labelled with two objects, a delay range $[d, D]$ with integer bounds $(0 \leq d \leq D)$, and ε, an integer (usually non-negative) described in Section 2.3. The set of events, E', always contains

a unique event, *root*, which is used to specify the startup behavior of the system. For the example in Figure 1, which corresponds to the three-process example in the introduction, we have

$$\begin{aligned} E' &= \{root, a, b\} \text{ and} \\ R' &= \{root \overset{[0,0],0}{\longmapsto} a,\ a \overset{[4,10],1}{\longmapsto} a,\ a \overset{[1,2],0}{\longmapsto} b,\ b \overset{[1,6],1}{\longmapsto} a,\ b \overset{[5,20],1}{\longmapsto} b\}\,. \end{aligned}$$

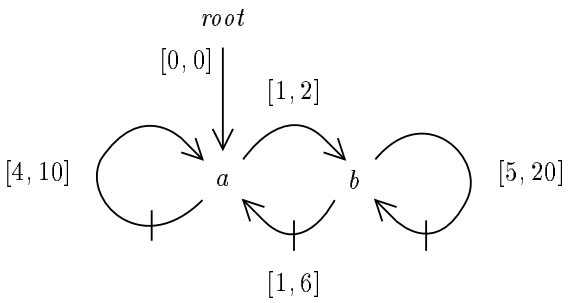

Figure 1: A process graph with three events, *root*, a and b. The number of lines drawn through an edge indicates the value of the occurrence index offset, ε, for the edge (e.g., $b \overset{[1,6],1}{\longmapsto} a$ constrains a_k relative to b_{k-1}).

We restrict our analysis to *well-formed* graphs, that is, graphs that are connected and have $\varepsilon(c) > 0$ for all cycles c in the graph, where $\varepsilon(c)$ is the sum of the ε values for all edges in the cycle c. Furthermore, *root* must have zero indegree, and for all other events v, there must be a path from *root* to v, i.e., only *root* has indegree of zero.

2.2 The Unfolded Process Graph

Consider the process graph $G' = \langle E', R' \rangle$. We denote the k^{th} occurrence of event $v \in E'$ as v_k, and refer to k as the *occurrence index* of v_k. We can represent each of the iterations of the process graph explicitly by *unfolding* the process graph. Let E be the set of all event occurrences (infinite in one direction) and let R be the set of rules generated by instantiating each rule template of R' for each of the event occurrences in E. We call the infinite directed graph constructed from the vertex set E and the edge set R the *unfolded process graph*, denoted by $G = \langle E, R \rangle$. G is a directed acyclic graph (DAG) that represents the infinite execution of the process graph, G'. The set of event occurrences E can be defined recursively from the basis clause $E = \{root_0\}$ and the inductive clause

$$\text{if } u_k \in E \text{ and } u \overset{[d,D],\varepsilon}{\longmapsto} v \in R', \text{ then } v_{k+\varepsilon} \in E\,.$$

The set of edges in the unfolded process graph R is defined as:

$$R = \left\{ u_k \overset{[d,D]}{\longmapsto} v_{k+\varepsilon} \;\middle|\; u \overset{[d,D],\varepsilon}{\longmapsto} v \in R' \text{ and } u_k, v_{k+\varepsilon} \in E \right\}\,.$$

Figure 2 shows a portion of the unfolded process graph for the example in Figure 1.

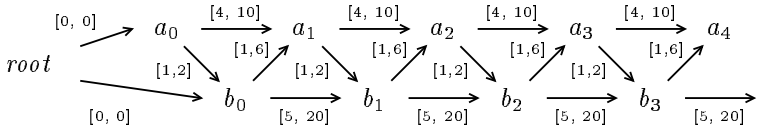

Figure 2: A portion of the unfolded process graph for the process graph in Figure 1.

Events can be characterized as *repeatable* or *non-repeatable* corresponding to whether there is a cycle in the process graph containing the event. The event *root* is non-repeatable as it has indegree of zero. A non-repeatable event will occur at most once in the unfolded process graph while a repeatable event occurs an infinite number of times. Non-repeating events arise because a non-trivial DAG can be used for specifying the startup behavior. For convenience, we drop the occurrence index (zero) when referring to occurrences of non-repeating events (e.g., we write *root* instead of $root_0$).

2.3 Execution Model

An *execution* of a process graph is the consistent assignment of time values to event occurrences. A *timing assignment*, τ, maps event occurrences to global time, thus $\tau(v_k)$ is the time of the k^{th} occurrence of event v. The delay information in R restricts the set of possible timing assignments. Formally, we define constraints on the time values introduced by each event occurrence, i.e., a consistent timing assignment satisfies:

$$\max\left\{\tau(u_{k-\varepsilon}) + d \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R\right\} \leq \tau(v_k) \leq \max\left\{\tau(u_{k-\varepsilon}) + D \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R\right\}. \quad (1)$$

The constraints on $\tau(v_k)$ embody the underlying semantics of a process graph's execution, i.e., an event can occur only when all of its incident events have occurred. Each incident event is delayed by some number in a bounded interval ($[d, D]$). Thus, the earliest time at which v_k can occur is constrained by d values, the latest by D values. Figure 3 shows a possible timing assignment for the graph in Figure 2 obtained by choosing the upper bound, D, for all delays.

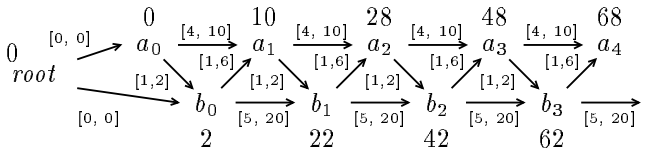

Figure 3: A portion of the unfolded process graph for the process graph in Figure 1. The vertices are annotated with the timing assignment $\tau(v_k)$ obtained by choosing the upper bound, D, for all delays.

2.4 Problem Definition

The problem we address in this paper is: given two events, s and t in E', and a separation in occurrence index β, determine the largest δ and the smallest Δ such that $\forall k \geq \max(0, \beta)$:

$$\delta \leq \tau(t_k) - \tau(s_{k-\beta}) \leq \Delta\,.$$

We address only the problem of finding the maximum separation, since the minimum separation, δ, can be obtained from a maximum separation analysis of $\forall k \geq \max(0, -\beta)$:

$$-\Delta \leq \tau(s_k) - \tau(t_{k-(-\beta)}) \leq -\delta\,.$$

To determine the bounds on the time separation between two consecutive a events, we would set $s = t = a$ and $\beta = 1$, and consider the bounds on $\tau(a_k) - \tau(a_{k-1})$. The timing assignment in Figure 3 indicates that the separation between two consecutive a events for the process graph in Figure 1 is between 10 and 20, i.e., that $10 \leq \tau(a_k) - \tau(a_{k-1}) \leq 20$. However, it turns out that there exists a timing assignment such that $\tau(a_k) - \tau(a_{k-1}) = 4$ (for some k) and another assignment such that $\tau(a_k) - \tau(a_{k-1}) = 25$. These are the extreme cases and thus the tightest bounds are $\delta = 4$ and $\Delta = 25$.

The following two sections describe the theory and implementation of an efficient algorithm for determining the maximum separation, i.e., for finding Δ. However, the reader may want to skip ahead to Section 5 for some applications of this kind of timing analysis before examining the details of the Time Separation of Events (TSE) algorithm.

3 Algorithm for an Acyclic Graph

Our algorithm for analyzing a process graph is based on an algorithm that determines the maximum separation in an *acyclic* graph [24], i.e., for a finite portion of the unfolded process graph. In Section 4 we will generalize this algorithm for infinite unfolded graphs.

Consider a particular event occurrence of the event t, t_α. Let Δ_α be the strongest bound for the separation problem given the occurrence index α, i.e.,

$$\tau(t_\alpha) - \tau(s_{\alpha-\beta}) \leq \Delta_\alpha\,.$$

We can determine Δ_α from a finite portion of the unfolded process graph created by only including the vertices for which there is a path to either t_α or $s_{\alpha-\beta}$. Name the resulting graph G_α. The algorithm consists of two simple phases. We first compute $m(v_k)$, the longest path from v_k to $s_{\alpha-\beta}$ using the lower delay bounds of the edges:

$$m(v_k) = \max\left\{ d(h) \;\middle|\; \text{all paths } v_k \overset{h}{\rightsquigarrow} s_{\alpha-\beta} \right\}\,,$$

where $d(h)$ is sum of the d values of the edges on the path h. We can compute the m-values by a reverse topological traversal starting from $s_{\alpha-\beta}$. If there is no path from v_k to $s_{\alpha-\beta}$, denoted by $v_k \not\rightsquigarrow s_{\alpha-\beta}$, we can assign an arbitrary constant to $m(v_k)$. Normally, we use $m(v_k) = 0$ although it sometimes is advantageous to choose different constants for the various v_k—see Section 4.7.

Second, we compute M-values using the D values by assigning $M(root) = 0$ and then for all other occurrences in (normal) topological order:

$$M(v_k) = \max\left\{ X_r \;\middle|\; r = u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k, r \in G_\alpha \right\} \tag{2}$$

where

$$X_r = \begin{cases} \min(0, M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k)) & \text{if } v_k \rightsquigarrow s_{\alpha-\beta} \\ M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k) & \text{if } v_k \not\rightsquigarrow s_{\alpha-\beta} \end{cases}$$

Theorem 1

$\Delta_\alpha = M(t_\alpha) - m(t_\alpha)$ is an achievable upper bound on the time separation between $s_{\alpha-\beta}$ and t_α.

Proof: (Sketch, full proof in Appendix A.) The proof consists of two parts. We first prove that for any consistent timing assignment τ, we have

$$\tau(t_\alpha) - \tau(s_{\alpha-\beta}) \leq M(t_\alpha) - m(t_\alpha)\,.$$

Then we show that the class of timing assignments

$$\tau(v_k) = M(v_k) - m(v_k) + c\,,$$

where c is an arbitrary constant, correspond to legal executions, and

$$\tau(t_\alpha) - \tau(s_{\alpha-\beta}) = M(t_\alpha) - m(t_\alpha)\,.$$

□

ACYCLIC-TSE$(G_\alpha, s_{\alpha-\beta}, t_\alpha)$

1 **for** u_j in reverse topological order of G_α

2 $$m(u_j) \leftarrow \begin{cases} 0 & \text{if } u_j = s_{\alpha-\beta} \\ \text{An arbitrary constant} & \text{if } u_j \not\rightsquigarrow s_{\alpha-\beta} \\ \max\left\{ m(v_k) + d \;\middle|\; u_j \overset{[d,D]}{\longmapsto} v_k \in G_\alpha \text{ and } v_k \rightsquigarrow s_{\alpha-\beta} \right\} & \text{if } u_j \rightsquigarrow s_{\alpha-\beta} \end{cases}$$

3 $M(root) \leftarrow 0$

4 **for** v_k in normal topological order of G_α

5 **if** $v_k \rightsquigarrow s_{\alpha-\beta}$ **then**

6 $M(v_k) \leftarrow \max\left\{ \min(0, M(u_j) + D - m(u_j) + m(v_k)) \;\middle|\; u_j \overset{[d,D]}{\longmapsto} v_k \in G_\alpha \right\}$

7 **else**

8 $M(v_k) \leftarrow \max\left\{ M(u_j) + D - m(u_j) + m(v_k) \;\middle|\; u_j \overset{[d,D]}{\longmapsto} v_k \in G_\alpha \right\}$

9 **return** $M(t_\alpha) - m(t_\alpha)$

Figure 4: Algorithm for determining the maximum separation between events occurrences $s_{\alpha-\beta}$ and t_α. G_α is a finite acyclic graph, i.e., a finite portion of the unfolded process graph.

The pseudo-code for the acyclic algorithm is shown in Figure 4. Informally, the algorithm works as follows: To maximize the value of $\tau(t_\alpha) - \tau(s_{\alpha-\beta})$ we need to find an execution that maximizes $\tau(t_\alpha)$ and minimizes $\tau(s_{\alpha-\beta})$. In the first pass the algorithm determines the minimum separation from any event to $s_{\alpha-\beta}$. In the second pass, events are delayed as much as possible using D-values. However, the delay for a given edge can not be assigned both d and D, and this is ensured by the minimization with zero in line 8.

Applying the algorithm to the example in Figure 1 (see Figure 5 for the computation of Δ_2) yields the following maximum separations:

$\tau(a_k) - \tau(a_{k-1}) \leq \Delta_k$

Δ_1	Δ_2	Δ_3	$\Delta_{>3}$
10	24	25	25

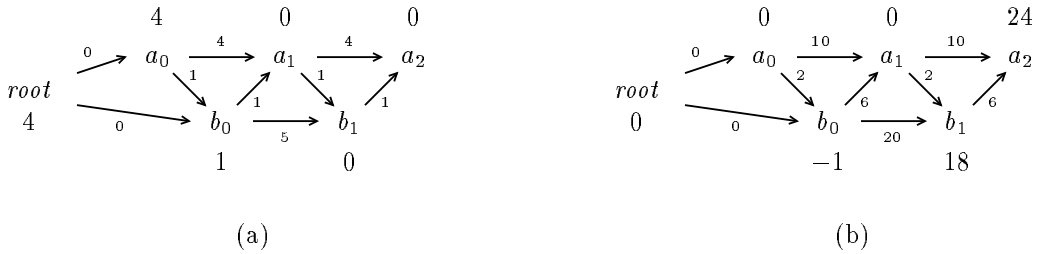

Figure 5: Finite acyclic graph, G_2, for obtaining Δ_2 for the process graph in Figure 1 given the parameters $s = t = a$ and $\beta = 1$, i.e., $s_{\alpha-\beta} = a_1$ and $t_\alpha = a_2$. In (a), the edges are labeled with the d values, and the vertices are labelled with the m-values obtained in the first phase of the algorithm. In (b), the edges are labeled with the D values, and the vertices are labelled with the M-values obtained in the second phase. We obtain $\Delta_2 = M(a_2) - m(a_2) = 24 - 0 = 24$.

To compute Δ, the maximum separation in time over all occurrences of s and t, separated in occurrence index by β, we maximize Δ_k over all values of k:

$$\Delta = \max\left\{\Delta_k \;\middle|\; k \geq \max(0, \beta)\right\} .$$

For the example in Figure 1 we thus have $\Delta = \max\{10, 24, 25, 25, \ldots\} = 25$. The problem, of course, is that this requires an infinite number of applications of the algorithm.

Before we present an algebraic solution that allows us to analyze the infinite unfolded graph, we present two additional examples constructed to illustrate both the simplicity and the complexity of the required analysis. Section 5 presents realistic applications of the timing analysis.

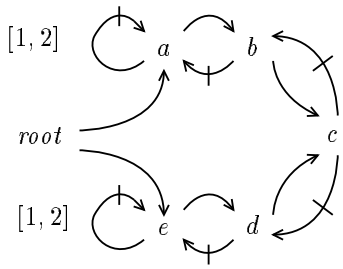

Figure 6: A process graph that represents two coupled pipelines. All unspecified delay ranges are $[0, 0]$.

The first example, in Figure 6, is a process graph that represents two coupled pipelines. If the pipelines were not coupled at c, the maximum separation between a and e would be unbounded. This is because the first pipeline (choosing a delay of 2 units between consecutive a occurrences) could be arbitrarily slower than the second pipeline (choosing the delay of 1 unit between consecutive e occurrences). The coupling of the pipelines forces one pipeline to wait for the other if it gets too

far ahead. For all $k \geq 0$, it can be shown that $\tau(a_k) - \tau(e_k) \leq 4$:

Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	$\Delta_{>4}$
0	1	2	3	4	4

This arises because we can have $\tau(a_0) = 0$, $\tau(a_1) = 2$, $\tau(a_2) = 4$, $\tau(a_3) = 6$, $\tau(a_4) = 8$, $\tau(a_5) = 10$ along with $\tau(e_0) = 0$, $\tau(e_1) = 1$, $\tau(e_2) = 2$, $\tau(e_3) = 3$, $\tau(e_4) = 4$, but we cannot have $\tau(e_5) = 5$ because of the dependency requiring $\tau(e_5)$ to occur no earlier than $\tau(a_3) = 6$ (the path $a \overset{[0,0],0}{\longmapsto} b \overset{[0,0],0}{\longmapsto} c \overset{[0,0],1}{\longmapsto} d \overset{[0,0],1}{\longmapsto} e$). Adding more stages to both pipelines (before the synchronization, c) would allow e to get further ahead of a.

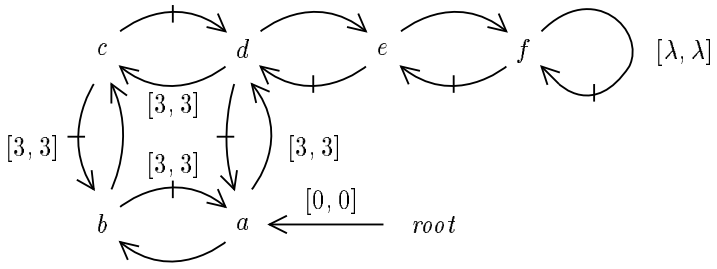

Figure 7: A process graph with unusual timing behavior. All unspecified delay ranges are $[1, 1]$.

Our second example, in Figure 7, exhibits interesting behavior. If $\lambda = 6$ then $\tau(a_k) - \tau(a_{k-1}) \leq 8$:

Δ_1	Δ_2	Δ_3	Δ_4	Δ_{odd}	Δ_{even}
4	8	4	8	4	8

If we change $\lambda = 9$ then $\tau(a_k) - \tau(a_{k-1}) \leq 9$:

Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	$\Delta_{>8}$
4	8	4	8	4	8	8	9	9

As the process graph is a repetitive system, presumably the Δ_k values will eventually reach a steady state, for example, $\Delta_{k+1} = \Delta_k$ for large k. Unfortunately, as the last example illustrates, the behavior of the Δ_k values can be non-monotonic and periodic, and might even start out periodic and then later stabilize to a constant value. In light of these examples, we believe no simple termination criteria exists for this approach.

4 The TSE Algorithm

Our solution to the problem is based on a structural decomposition of the unfolded process graph that exploits its repetitive nature. By dividing the unfolded process graph up into segments and representing the computation of the finite graph algorithm in a symbolic manner we can reuse the computations for each segment.

4.1 Introducing Functions

We introduce a symbolic execution of the acyclic algorithm in Figure 4. Instead of computing the numeric M-values in (2), we compute functions that relate M-values with one another. We introduce the algebraic structure[1] $(\mathcal{F}, \oplus, \otimes, \overline{0}, \overline{1})$. Each element in $\mathcal{F}$ is a piecewise-linear, monotonicly non-decreasing function represented by a set of pairs. The singleton set, $\{\langle l, w\rangle\}$, represents the function $f(x) = \min(x + l, w)$. In general, the set

$$\{\langle l_1, w_1\rangle, \langle l_2, w_2\rangle, \ldots, \langle l_n, w_n\rangle\} \tag{3}$$

corresponds to the function

$$f(x) = \max\{\min(x + l_i, w_i) \mid 1 \leq i \leq n\}\,. \tag{4}$$

We associate two binary operators with functions: function maximization, $f \oplus g$, and function composition, $f \otimes g$. It follows from (4) that function maximization is defined as set union: $f \oplus g = f \cup g$. Function composition, $f = g \otimes h$, is defined as $f(x) = h(g(x))$. Notice that we use left-to-right function composition [14]. For $g = \{\langle l_1, w_1\rangle\}$ and $h = \{\langle l_2, w_2\rangle\}$ we have

$$\begin{aligned}(g \otimes h)(x) &= h(g(x)) = \min(g(x) + l_2, w_2)\\ &= \min(\min(x + l_1, w_1) + l_2, w_2)\\ &= \min(x + l_1 + l_2, \min(w_1 + l_2, w_2))\\ &= \{\langle l_1 + l_2, \min(w_1 + l_2, w_2)\rangle\}\,.\end{aligned} \tag{5}$$

In general, let

$$\begin{aligned}g &= g_1 \oplus \cdots \oplus g_n \quad \text{and}\\ h &= h_1 \oplus \cdots \oplus h_m\,,\end{aligned}$$

where g_i and h_i are singleton sets. Function composition is performed using distributivity:

$$g \otimes h = \bigoplus \left\{ g_i \otimes h_j \;\middle|\; 1 \leq i \leq n, 1 \leq j \leq m \right\}.$$

The elements $\overline{0}$ and $\overline{1}$ are the identity elements for function maximization and composition, respectively. We choose $\overline{0} = \{\}$ and $\overline{1} = \{\langle 0, \infty\rangle\}$. Note that $\overline{0}$ is an annihilator for function composition, i.e., $f \otimes \overline{0} = \overline{0} \otimes f = \overline{0}$ for all $f \in \mathcal{F}$ (even when f is a constant function). The algebraic structure $(\mathcal{F}, \oplus, \otimes, \overline{0}, \overline{1})$ forms a *closed semiring*, that is, $(\mathcal{F}, \oplus, \overline{0})$ and $(\mathcal{F}, \otimes, \overline{1})$ are monoids (i.e., are closed, associative, and have an identity), $\overline{0}$ is an annihilator (i.e., $a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$), $\oplus$ is commutative, and $\otimes$ distributes over $\oplus$ (including infinite summaries). Except for distributivity, these properties are trivially satisfied. Distributivity relies on the monotonicly non-decreasing nature of the functions in $\mathcal{F}$. Notice that function composition ($\otimes$) is not commutative.

[1] Similar to a $(\max, +)$-algebra [9, 4]. The main differences are that the elements of $\mathcal{F}$ are functions instead of numbers, the max-operator maximizes functions, and the $+$-operator composes functions.

4.2 Functional Formulation of Acyclic Algorithm

We can now express phase 2 of the acyclic algorithm in terms of functions. We associate a function, f_r, with each edge $u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k$ in the unfolded process graph:

$$f_r = \begin{cases} \{\langle l_r, 0\rangle\} & \text{if } v_k \rightsquigarrow s_{\alpha-\beta} \\ \{\langle l_r, \infty\rangle\} & \text{if } v_k \not\rightsquigarrow s_{\alpha-\beta} \end{cases} \tag{6}$$

where $l_r = D - m(u_{k-\varepsilon}) + m(v_k)$. Let $u_{k-\varepsilon} \overset{f_r}{\longmapsto} v_k$ denote this association. Notice that $f_r(M(u_{k-\varepsilon}))$ is equal to X_r in (2).

Using function composition and function maximization, we can create a function f that relates $M(root)$ to $M(v_k)$, i.e., $M(v_k) = f(M(root))$. Let $F_{u_j \mapsto v_k}$ denote the function relating $M(u_j)$ to $M(v_k)$. For all event occurrences, v_k, $F_{v_k \mapsto v_k}$ is the identity function, $\overline{1}$. In general, the function $F_{root \mapsto v_k}$ is defined recursively as

$$F_{root \mapsto v_k} = \bigoplus \left\{ F_{root \mapsto u_{k-\varepsilon}} \otimes f_r \;\middle|\; u_{k-\varepsilon} \overset{f_r}{\longmapsto} v_k \in R \right\} . \tag{7}$$

The separation between $s_{\alpha-\beta}$ and t_α is $\Delta_\alpha = M(t_\alpha) - m(t_\alpha)$ where $M(t_\alpha) = F_{root \mapsto v_k}(M(root)) = F_{root \mapsto v_k}(0)$.

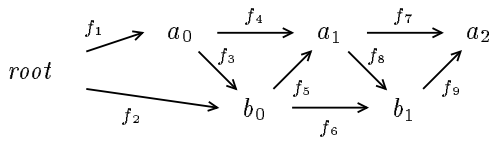

Figure 8: Fragment of unfolded process graph annotated with functions corresponding to each edge (the m-values are given in Figure 5 (a)).

For the example in Figure 5 (see Figure 8), we relate $M(root)$ to $M(b_0)$ with the function $F_{root \mapsto b_0} = f_1 \otimes f_3 \oplus f_2 = \{\langle 0, 0\rangle\} \otimes \{\langle -1, 0\rangle\} \oplus \{\langle -3, 0\rangle\} = \{\langle -1, -1\rangle, \langle -3, 0\rangle\}$. Evaluating the function at $M(root) = 0$ yields -1, which is the value obtained for $M(b_0)$ in Figure 5 (b). The functions $F_{root \mapsto b_0}$ and $F_{root \mapsto a_0}$ are then used to relate $M(root)$ to $M(a_1)$, etc., until a function that relates $M(root)$ to $M(t_\alpha)$ is created. In our example, $t_\alpha = a_2$ and the construction produces $F_{root \mapsto a_2} = \{\langle 22, 25\rangle, \langle 24, 24\rangle\}$. We get $\Delta_2 = F_{root \mapsto a_2}(0) - 0 = 24$, where $F_{root \mapsto a_2}$ is evaluated according to (4).

4.3 Incrementally Computing Δ_k Values

The algorithm for an acyclic graph (see Figure 4) can be used to determine Δ_k for a specific value of k. However, we are interested in the maximum value of Δ_k over all k. The functional representation of the acyclic algorithm allows us to efficiently compute Δ_k for increasing k-values by reusing the functions.

Assume we have constructed the function $F_{root \mapsto t_\alpha}$ for the graph G_k. Note that because functions are associative, they can be composed in any order. Following (7), we can construct $F_{root \mapsto t_\alpha}$ either starting from *root* going forward in the graph, or starting from t_α going backwards. Starting from the *root* node causes the function $F_{root \mapsto v_k}$ to be constructed for each node v_k. On the other hand, starting from t_α causes the function $F_{v_k \mapsto t_\alpha}$ to be constructed.

Because m-values are compute backward from $s_{\alpha-\beta}$ and each f_r depends on the m-values at the event occurrences named in the rule r, we can reuse functions only if we apply the *backward* method. This motivates a change in the numbering of event occurrences. Instead of numbering events starting with zero and increasing as the process graph is unfolded (forward), events are numbered relative to a reference event. We use t_α as the reference event and let the *relative occurrence index* for node v_k to be $\alpha - k$, written $v_{(k)}$. By definition, the relative occurrence index for t_α is 0. Relative occurrence indices are positive for nodes topologically left of t_α in the unfolded process graph and negative for nodes topologically right of t_α. Figure 9 shows the graph segment from Figure 8 using relative occurrences indices.

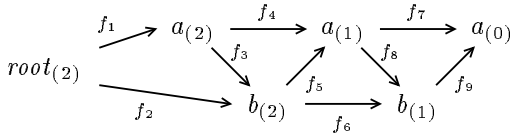

Figure 9: Fragment of unfolded process graph using relative occurrence indices.

A finite portion of the unfolded process graph including only event occurrences with relative occurrence index no larger than k is denoted by $G_{(k)}$. The graph in Figure 9 is denoted $G_{(2)}$. We can find Δ_k from the function $F_{root_{(k)} \mapsto t_{(0)}}$ in the graph $G_{(k)}$, since $\Delta_k = F_{root_{(k)} \mapsto t_{(0)}}(0) - m(t_{(0)})$.

Now consider finding the next separation value, Δ_{k+1}. Instead of starting over from scratch, rebuilding the graph $G_{(k+1)}$ and then forming the function for this graph, we choose to reuse the work done in the computation of Δ_k and instead extend $G_{(k)}$ by the addition of $|E'|$ new event occurrences (and $|R'|$ new rules) with relative occurrence index $k+1$. This is illustrated in Figure 10. Functions for the nodes in $G_{(k+1)}$ that are also in $G_{(k)}$ are unchanged since the

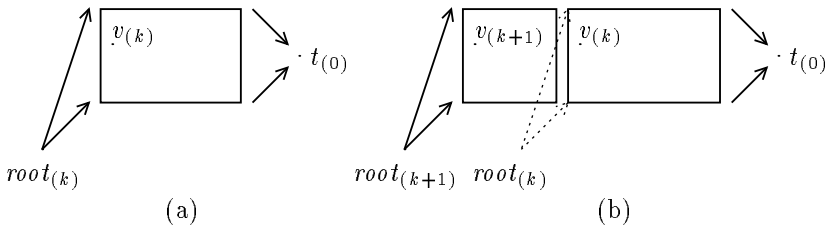

Figure 10: To the left a generic graph, $G_{(k)}$. To the right the graph $G_{(k+1)}$.

m-values for these nodes have not changed. Computing the functions $F_{v_{(k+1)} \mapsto t_\alpha}$ for the new $v_{(k+1)}$ nodes can accomplished using $O(|R'|)$ scalar function operations. This results in an efficient one pass algorithm for computing values for Δ_k for increasing values of k. Figure 11 shows $G_{(k)}$ for the example in Figure 1 for $k = 1, 2, 3$ and the corresponding functions for the added nodes.

This incremental way of computing Δ_k leads to a partially correct algorithm for computing Δ, the maximum separation over all Δ_k values. Compute the Δ_k values starting from the graph $G_{(\max(0,\beta))}$ and maximize over Δ_k values for increasing k. Bounds on Δ from above and from below can be determined for each iteration. If the bounds converge, we can stop and report the result. The algorithm is only partially correct because the derived bounds may not converge. This is discussed below.

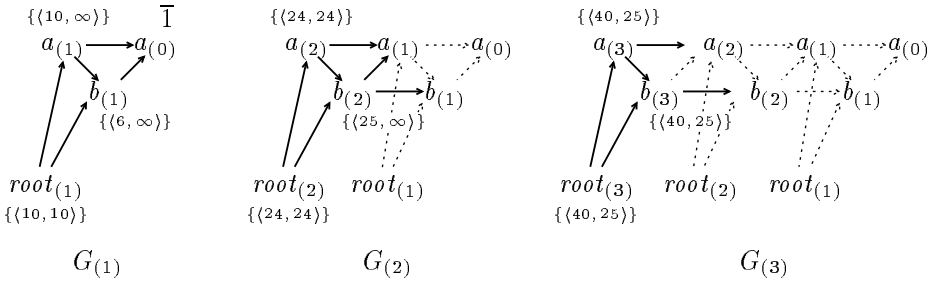

Figure 11: The graphs $G_{(1)}$, $G_{(2)}$, and $G_{(3)}$ and corresponding functions for computing $\Delta_1 = 10$, $\Delta_2 = 24$, and $\Delta_3 = 25$, respectively. Only the solid edges are added in each iteration.

Computing a lower bound on Δ, $\Delta^{\perp}$, is straightforward. Because Δ is the maximum of Δ_k values over all $k \geq \max(0, \beta)\}$, the maximum of any subset of Δ_k values is a lower bound on Δ.

In order to compute an upper bound on Δ we need to introduce a *cut* of the graph $G_{(k)}$. The cut is defined by a *cutset*, a set of relative event occurrences (members of $G_{(k)}$) such that for each further unfolding, $G_{(j)}$, $j \geq k$, every path from $root_{(j)}$ to $t_{(0)}$ goes through an element of the cutset.

For a given cutset X for $G_{(k)}$, the function from $root_{(k)}$ to $t_{(0)}$ can be expressed as

$$\bigoplus \left\{ F_{root_{(k)} \mapsto u_j} \otimes F_{u_j \mapsto t_{(0)}} \;\middle|\; u_{(j)} \in X \right\}. \tag{8}$$

For further unfoldings of the process graph, only the functions to the left of the "$\otimes$" symbol in (8) change. By replacing the functions $F_{root_{(k)} \mapsto u_j}$ with "worst case" functions, we can compute an upper bound on Δ, $\Delta^{\top}$. The following lemma is used to bound the codomain of functions.

Lemma 2

Let f be a function constructed for a sub-graph where all nodes have a path to $s_{(\beta)}$. Then $\forall x : f(x) \leq 0$.

Proof: As all nodes have a path to $s_{(\beta)}$, f is constructed from pairs of the form $\langle l, 0 \rangle$, where l is any integer. From (5) it follows that all pairs in f, $\langle l, w \rangle$, have $w \leq 0$ (from a simple induction on the number of compositions). The result then follows from (4). □

By evaluating the functions $F_{v_k \mapsto t_{(0)}}$ at 0, i.e., replacing $F_{root \mapsto v_k}$ with 0 in (8), we obtain an upper bound on Δ:

$$\Delta^{\top} = \max \left\{ F_{v_k \mapsto t_{(0)}}(0) \;\middle|\; v_{(k)} \in X \right\} - m(t_{(0)}). \tag{9}$$

The upper bound may improve when further unfolding the graph and choosing a new cutset X that separates the graph so that more nodes are included to the right of the cut. Note that only when all nodes $v_{(k)}$ topologically left of the nodes in X have a path to $s_{(\beta)}$ does (9) compute an upper bound on Δ. If $v_{(k)} \not\rightarrow s_{(\beta)}$, the functions left of X may not be constructed from pairs of the form $\langle l, 0 \rangle$ (but rather of pairs of the form $\langle l, \infty \rangle$) and Lemma 2 does not apply.

Each time a new Δ_k value is computed the bounds are updated and if the bounds converge the algorithm terminates. For the example in Figure 11, the lower bound is 10, 24, and 25 for the three unfoldings, respectively. An upper bound is found from the cutset $X = \{a_{(1)}, b_{(1)}\}$. For $G_{(1)}$, no upper bound can be computed, i.e., $\Delta^{\top} = \infty$. For $G_{(2)}$, we choose $X = \{a_{(2)}, b_{(2)}\}$.

An upper bound is found by maximizing the function at the nodes in the cutset evaluated at 0: $\max(\min(24+0, 24), \min(25+0, \infty)) = 25$. This is also the upper bound for $G_{(3)}$. Thus, after three unfoldings the bounds converge and the maximum separation of 25 can be reported.

A subtle point is that the lower bound $\Delta^{\perp}$ can be larger than the upper bound $\Delta^{\top}$. The reason is that the bound $\Delta^{\top}$ for a given graph $G_{(k)}$ is a bound on all *further* unfoldings, i.e., $\forall j > k : \Delta_j \leq \Delta^{\top}$. But there may be some initial transient behavior that forces the first event occurrences to be separated by more than can be achieved later. For example, by changing the initial edge $root \overset{[0,0],0}{\longmapsto} b$ to $root \overset{[0,94],0}{\longmapsto} b$ for the process graph in Figure 1, we get $\Delta_1 = 100$, but $\forall j > 1 : \Delta_j = 25$. After one unfolding we have $\Delta^{\perp} = 100$ and $\Delta^{\top} = \Delta_{>1} = 25$.

```
UNFOLD(G, s, t, β, k_max)
 1  k ← max(0, β)
 2  Δ⊥ ← −∞
 3  Δ⊤ ← ∞
 4  while (Δ⊥ < Δ⊤ ∧ k < k_max) {
 5     Construct G_(k)
 6     Δ_k ← F_{root_(k) ↦ t_(0)}(0) − m(t_(0))
 6     X ← CUTSET[G_(k)]
 7     Δ⊥ ← max(Δ⊥, Δ_k)
 8     Δ⊤ ← max{F_{v_k ↦ t_(0)}(0) | v_(k) ∈ X} − m(t_(0))
 9     k ← k + 1
10  }
11  return(Δ⊥, Δ⊤, X)
```

Figure 12: Algorithm for unfolding the process graph and computing bounds on Δ.

The pseudo-code for the procedure UNFOLD is shown in Figure 12. The process graph is unfolded until either the bounds converge or an upper limit on the number of unfoldings, k_{max}, is reached. For each unfolding, the graph is extended with $O(|E'|)$ new nodes and their functions. Given a cutset X for the graph, the bounds on Δ are determined. This approach for finding the maximum separation between two events is efficient on some examples [25], but has two major drawbacks.

First, we know of no necessary condition for determining when to stop unfolding the graph and report the result, i.e., the bounds derived above may not converge. A simple example is shown in Figure 13. The two startup rules determine the maximum separation between a_k and e_k to be 0 for all $k \geq 0$. This example demonstrates that the startup rules can determine the maximum separation at every point in the infinite execution. The start-up rules are ignored when computing the upper bound and in this case the upper bound will never converge to the correct result.

Second, even if a necessary termination condition is developed, the approach may be inefficient. There are examples where it is necessary to unfold the process graph many times before the bounds converge and the necessary number of unfoldings depend on the delay values in the process graph. The process graph in Figure 14 requires 149 unfoldings before the bounds converge and the number of unfoldings can be made arbitrarily large by changing the delay values of the process graph.

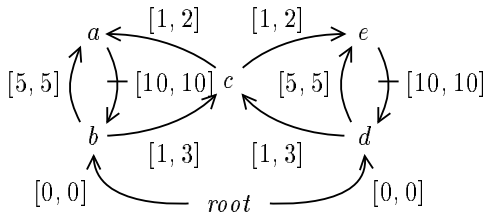

Figure 13: Two processes synchronizing at c.

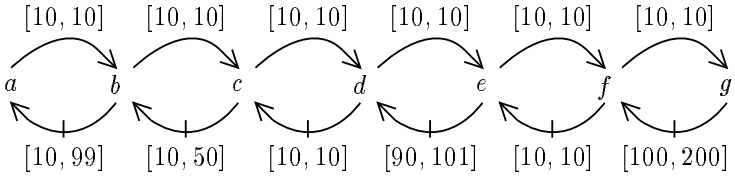

Figure 14: A 6-stage pipeline that needs to be unfolded 149 times before the bounds converge for the separation analysis $\tau(b_k) - \tau(d_{k-2})$. The startup rules, $root \overset{[0,0],0}{\longmapsto} a$, $root \overset{[210,210],0}{\longmapsto} e$, $root \overset{[310,310],0}{\longmapsto} g$, have been left out for clarity.

We now consider the structure of the m-values. It turns out that after a number of unfoldings they enter a regular and repetitive pattern. This can be used to implicitly analyze the infinite unfolded process graph, leading to an efficient algorithm that addresses the two problems just described.

4.4 Repetition of the m-values

Recall that $m(v_k)$ is the longest path $v_{(k)} \rightsquigarrow s_{(\beta)}$ using the lower delay bound on the edges, d. Since the m-values are constructed from a repetitive system (the process graph) the values eventually are determined by the *maximum ratio cycles* in the process graph [19]. A maximum ratio cycle c is a cycle with ratio $d(c)/\varepsilon(c)$ equal to that of the maximum ratio r:

$$r = \max\left\{ \frac{d(c)}{\varepsilon(c)} \;\middle|\; c \text{ a simple cycle in } G' \right\} .$$

Intuitively, when the m-values for all event occurrences are determined repetitively using using maximum ratio cycles, we say the m-values *repeat*. Formally, for a strongly connected process graph there exists integers $k^\star$ and $\varepsilon^\star$ such that for all $k \geq k^\star + \beta$ and all $v \in E'$

$$m(v_{(k+\varepsilon^\star)}) - m(v_{(k)}) = r\varepsilon^\star , \tag{10}$$

where $k^\star$ is the number of unfoldings of the process graph (backwards relative to $s_{\alpha-\beta}$) before all of the m-values repeat and $\varepsilon^\star$ is the occurrence period of this repetition.

Figure 15 illustrates the behavior of the m-values for the process graph in Figure 1. Both $k^\star$ and $\varepsilon^\star$ are values specific to a particular process graph. For example, changing the delays $[4, 10]$ and $[5, 20]$ to [999,1000] and [1000,1000], respectively, changes[2] $k^\star$ from 3 to 998.

Algorithms for computing $k^\star$, $\varepsilon^\star$, and r are described in Section B.2.

[2]Note that only the lower delay bounds affect $k^\star$.

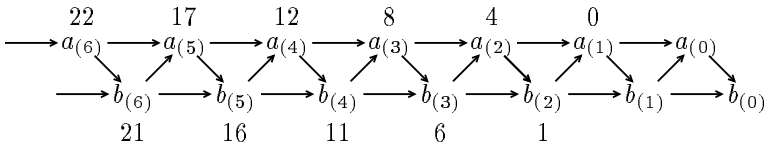

Figure 15: A portion of the unfolded process graph for the process graph in Figure 1 labeled with m-values ($s_{(\beta)} = a_{(1)}$). The m-values repeat when $m(a_{(4)}) - m(a_{(3)}) = r\varepsilon^\star = 5$ which occurs after three unfoldings relative to $a_{(1)}$, thus $k^\star = 3$. The occurrence period of the repetition is one, i.e., $\varepsilon^\star = 1$.

4.5 Introducing Matrices

The notion of a cutset was introduced in computing an upper bound on Δ. We now expand on the use of cutsets. A node $v_{(k)}$ is *left of* (or in) the cutset X, denoted $v_{(k)} \preceq X$, if

$$\exists u_{(j)} \in X : v_{(k)} \rightsquigarrow u_{(j)} \text{ or } v_{(k)} \in X\,.$$

A cutset X is left of (or equal to) the cutset Y, $X \preceq Y$, if

$$\forall v_{(k)} \in X : v_{(k)} \preceq Y\,.$$

Finally, $X_{\ll\omega}$ denotes a cutset X *shifted to the left* by ω:

$$X_{\ll\omega} = \left\{ v_{(k+\omega)} \;\middle|\; v_{(k)} \in X \right\}.$$

Now consider two cutsets for the graph $G_{(k)}$, X and Y. We overload F to denote a *matrix* of functions: $F_{X\mapsto Y}$ denotes the $|X|\times|Y|$ matrix containing functions relating M-values at nodes in X to M-values at nodes in Y. Using $(\oplus,\otimes)$ matrix multiplication, that is, function maximization for scalar addition, and function composition for scalar multiplication, we can decompose $F_{root_{(k)}\mapsto t_{(0)}}$ as

$$F_{root_{(k)}\mapsto t_{(0)}} = F_{root_{(k)}\mapsto X}\, F_{X\mapsto Y}\, F_{Y\mapsto t_{(0)}}\,,$$

where $F_{root_{(k)}\mapsto X}$ is a row-vector, $F_{X\mapsto Y}$ is a $|X|\times|Y|$ matrix, and $F_{Y\mapsto t_{(0)}}$ is a column-vector. The result of multiplying the three matrices is a 1×1 matrix whose single element is the function $F_{root_{(k)}\mapsto t_{(0)}}$.

For the graph in Figure 9, a possible decomposition is $X = \{a_{(2)}, b_{(2)}\}$ and $Y = \{a_{(1)}, b_{(1)}\}$ yielding

$$F_{root_{(2)}\mapsto X}\, F_{X\mapsto Y}\, F_{Y\mapsto t_{(0)}} = \begin{pmatrix} f_1 & f_1\otimes f_3 \oplus f_2 \end{pmatrix} \begin{pmatrix} f_4 & f_4\otimes f_8 \\ f_5 & f_5\otimes f_8 \oplus f_6 \end{pmatrix} \begin{pmatrix} f_7 \\ f_9 \end{pmatrix}.$$

4.6 Putting it All Together

A key observation is that when the m-values repeat, i.e., (10) holds, the difference in m-values between any two nodes is the same as the difference for the same nodes $\varepsilon^\star$ occurrences further back in the unfolded process graph. Let X_0 denote a cutset such that the m-values for all nodes left of X_0 repeat. Then, for all edges $u_{(j)} \overset{[d,D]}{\longmapsto} v_{(k)}$, where $v_{(k)} \preceq X_0$, (10) implies that

$$m(u_{(j+\varepsilon^\star)}) - m(v_{(k+\varepsilon^\star)}) = m(u_{(j)}) - m(v_{(k)})\,. \tag{11}$$

From (6) and (11) it follows that the function for the edge $u_{(j+\varepsilon^\star)} \stackrel{[d,D]}{\longmapsto} v_{(k+\varepsilon^\star)}$ is the same as for the edge $u_{(j)} \stackrel{[d,D]}{\longmapsto} v_{(k)}$. Therefore, considering a segment defined by two cutsets, X and Y, the functions relating the M-values of the nodes in the cutsets are the same as those relating the M-values of nodes in the cutsets shifted to the left by any multiple of $\varepsilon^\star$:

$$\forall n \geq 0 : F_{X_{\ll n\varepsilon^\star} \mapsto Y_{\ll n\varepsilon^\star}} \equiv F_{X \mapsto Y} \tag{12}$$

as long as $Y \preceq X_0$. By choosing the cutsets X and Y appropriately, we can construct the functions for one segment and reuse this segment for later occurrences. Let $X \preceq X_0$ and let $\mathbf{T}$ be a $|X| \times 1$ matrix defined as

$$\mathbf{T} = F_{X \mapsto t_{(0)}} .$$

We let $\mathbf{R}_i$ denote a $1 \times |X|$ matrix defined as

$$\mathbf{R}_i = F_{root_{(k_0+i)} \mapsto X_{\ll i}} ,$$

where i is a non-negative integer parameter and k_0 is such that $\mathbf{R}_0\,\mathbf{T}$ is the function for the graph $G_{(k_0)}$. Finally, $\mathbf{S}_i$ denotes the square matrix

$$\mathbf{S}_i = F_{X_{\ll i+1} \mapsto X_{\ll i}} .$$

Thus, $\mathbf{R}_i$ represents the initial segment and the matrix $\mathbf{S}_i$ represents one "unfolding" of G', i.e., the functions for a portion of the graph defined by $X_{\ll i+1}$ and $X_{\ll i}$. We overload F and use $F_{[n]}$ to denote the function for the graph $G_{(k_0+n)}$ for $n \geq 0$, i.e.,

$$\begin{aligned} F_{[n]} &= F_{root_{(k_0+n)} \mapsto t_{(0)}} \\ &= \mathbf{R}_n\,\mathbf{S}_{n-1}\,\mathbf{S}_{n-2} \cdots \mathbf{S}_2\,\mathbf{S}_1\,\mathbf{S}_0\,\mathbf{T} . \end{aligned}$$

Figure 16 illustrates how $F_{[n]}$ for $n = 1, 2, 3$ is obtained by piecing together an $\mathbf{R}_i$ segment and

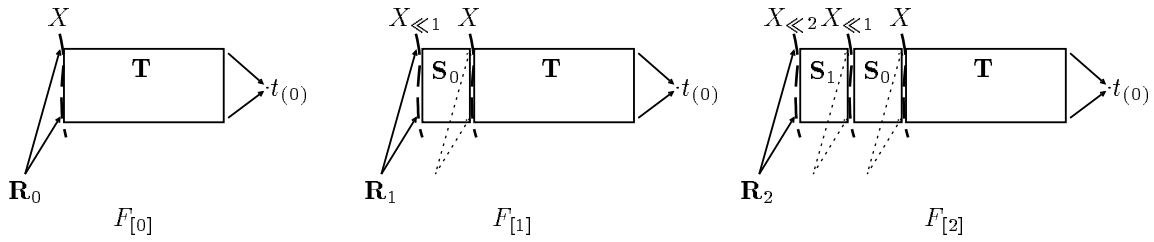

Figure 16: Construction of $F_{[n]}$ for increasing n. The function $F_{[n]}$ for $n \geq 0$ is constructed from $\mathbf{R}_n\,\mathbf{S}_{n-1}\,\mathbf{S}_{n-2} \cdots \mathbf{S}_1\,\mathbf{S}_0\,\mathbf{T}$. The figure shows the three graphs for the construction of $F_{[0]}$, $F_{[1]}$, and $F_{[2]}$.

multiple $\mathbf{S}_i$ segments. From (12) it follows that some of the $\mathbf{R}_i$ and $\mathbf{S}_i$ matrices are the same. In particular, all $\mathbf{S}_i$ matrices separated by a multiple of $\varepsilon^\star$ are identical:

$$\forall i \geq 0 : \mathbf{S}_i \equiv \mathbf{S}_{i \bmod \varepsilon^\star} ,$$

and this is also true of every $\mathbf{R}_i$ separated by $\varepsilon^\star$. This property can be used to find every $\varepsilon^\star$'th function efficiently, i.e., $F_{[n\varepsilon^\star]}$ for any $n \geq 0$ is computed as

$$F_{[n\varepsilon^\star]} = \mathbf{R}_{n\varepsilon^\star}\, \mathbf{S}_{n\varepsilon^\star-1}\, \mathbf{S}_{n\varepsilon^\star-2} \cdots \mathbf{S}_1\, \mathbf{S}_0\, \mathbf{T} = \mathbf{R}_0\, (\mathbf{S})^n\, \mathbf{T}\,, \tag{13}$$

where the square matrix $\mathbf{S}$ is defined as

$$\mathbf{S} = \mathbf{S}_{\varepsilon^\star-1}\, \mathbf{S}_{\varepsilon^\star-2} \cdots \mathbf{S}_1\, \mathbf{S}_0\,.$$

Note that if $\varepsilon^\star = 1$, the $\mathbf{S}_i$ matrices are independent of i and we get $\mathbf{S} = \mathbf{S}_0$. The maximum of $F_{[n\varepsilon^\star]}$ over all $n \geq 0$ is found directly from (13):

$$\bigoplus_{n\geq 0} F_{[n\varepsilon^\star]} = \mathbf{R}_0\, \mathbf{T} \oplus \mathbf{R}_0\, \mathbf{S}\, \mathbf{T} \oplus \mathbf{R}_0\, \mathbf{S}^2\, \mathbf{T} \oplus \mathbf{R}_0\, \mathbf{S}^3\, \mathbf{T} \oplus \mathbf{R}_0\, \mathbf{S}^4\, \mathbf{T} \oplus \cdots\,, \tag{14}$$

which by matrix algebra can be rewritten as

$$\mathbf{R}_0 \left(\mathbf{I} \oplus \mathbf{S} \oplus \mathbf{S}^2 \oplus \mathbf{S}^3 \oplus \mathbf{S}^4 \oplus \cdots\right) \mathbf{T}\,, \tag{15}$$

where $\mathbf{I}$ is the identity matrix. By defining $\mathbf{R}$ to be the maximum over $\varepsilon^\star$ unfoldings

$$\mathbf{R} = \bigoplus \left\{ \mathbf{R}_i \mathbf{S}_{i-1}\, \mathbf{S}_{i-2} \cdots \mathbf{S}_1\, \mathbf{S}_0 \;\middle|\; 0 \leq i < \varepsilon^\star \right\}\,,$$

we can find the maximum of $F_{[n]}$ over all $n \geq 0$ from

$$\bigoplus_{n\geq 0} F_{[n]} = \mathbf{R} \left(\mathbf{I} \oplus \mathbf{S} \oplus \mathbf{S}^2 \oplus \mathbf{S}^3 \oplus \mathbf{S}^4 \oplus \cdots\right) \mathbf{T} = \mathbf{R}\, \mathbf{S}^*\, \mathbf{T}\,, \tag{16}$$

where $\mathbf{S}^*$ is the *matrix closure* of $\mathbf{S}$. A matrix closure algorithm [1] can be used to compute $\mathbf{S}^*$ because $(\mathcal{F}, \oplus, \otimes, \overline{0}, \overline{1})$ forms a closed semiring. This is the key observation that allows us to implicitly analyze the infinite unfolded process graph. Evaluating the function computed by $\mathbf{R}\, \mathbf{S}^*\, \mathbf{T}$ at 0 and subtracting $m(t_{(0)})$ computes

$$\Delta_{\geq k_0} = \max \{\Delta_k \mid k \geq k_0\}\,.$$

The matrices $\mathbf{R}$ and $\mathbf{S}$ can be computed in a single sweep, unfolding the process graph $\varepsilon^\star$ times backwards, starting from $X \preceq X_0$. This is done by adding *root* to the cutset X. Then a single unfolding from $X_{\ll i}$ to $X_{\ll i+1}$ computes both $\mathbf{R}_i$ and $\mathbf{S}_i$ simultaneously:

$$F_{X_{\ll i+1} \mapsto X_{\ll i}} = \begin{pmatrix} \mathbf{S}_i & \overline{0} \\ \mathbf{R}_i & \overline{1} \end{pmatrix}.$$

As before, $\mathbf{S}_i$ relates the M-values at nodes in $X_{\ll i+1}$ (without the *root* node) to M-values at nodes in $X_{\ll i}$ (also without the *root* node). The $\overline{0}$-vector represents the fact that there are no paths from nodes in $X_{\ll i+1}$ to *root*. The row-vector $\mathbf{R}_i$ relates the *root* node to nodes in $X_{\ll i}$. Finally, setting the last position to $\overline{1}$ causes the product to maximize over the *root*-nodes. After $\varepsilon^\star$ unfoldings we get:

$$\bigotimes_{\varepsilon^\star > i \geq 0} \begin{pmatrix} \mathbf{S}_i & \overline{0} \\ \mathbf{R}_i & \overline{1} \end{pmatrix} = \begin{pmatrix} \mathbf{S}_{\varepsilon^\star-1}\, \mathbf{S}_{\varepsilon^\star-2} \cdots \mathbf{S}_1\, \mathbf{S}_0 & \overline{0} \\ \bigoplus \left\{ \mathbf{R}_i \mathbf{S}_{i-1}\, \mathbf{S}_{i-2} \cdots \mathbf{S}_1\, \mathbf{S}_0 \;\middle|\; 0 \leq i < \varepsilon^\star \right\} & \overline{1} \end{pmatrix} = \begin{pmatrix} \mathbf{S} & \overline{0} \\ \mathbf{R} & \overline{1} \end{pmatrix}. \tag{17}$$

Thus, unfolding $\varepsilon^\star$ times back we get exactly $\mathbf{R}$ and $\mathbf{S}$ which are needed to compute (16).

The pseudo-code for the TSE-algorithm is shown in Figure 17. Details of performing function operations and forming the matrix closure are described in Appendix B. Note that for a strongly connected process graph the TSE algorithm is guaranteed to terminate *and* find the tightest possible bound. The matrix closure in (16) implicitly performs an infinite analysis, thus producing an exact bound in finite time.

```
TSE(G', s, t, β)
 1  k* ← COMPUTE-k*[G', s]
 2  ε* ← COMPUTE-ε*[G', s]
 3  (Δ⊥, Δ⊤, X) ← UNFOLD[G', s, t, β, k* + β]
 4  if Δ⊥ ≥ Δ⊤ then return Δ⊥
 5  T ← F_{X ↦ t_(0)}
 6  for i ← 0, 1, ..., ε* − 1 {
 7     R_i ← F_{root_(k0+i) ↦ X≪i}
 8     S_i ← F_{X≪i+1 ↦ X≪i}
 9  }
10  ( S  0̄ )  ←  ⊗_{ε*>i≥0} ( S_i  0̄ )
    ( R  1̄ )               ( R_i  1̄ )
11  F ← R S* T
12  Δ_{≥k*+β} ← F(0) − m(t_(0))
13  return max(Δ_{≥k*+β}, Δ⊥)
```

Figure 17: Algorithm for determining the maximum separation between events s and t separated by β in occurrence index. G' is a well-formed strongly connected process graph.

4.7 Non-strongly Connected Process Graphs

The TSE algorithm requires the process graph G' to be strongly connected. Otherwise the m-values may not all eventually repeat. However, there are some classes of non-strongly connected process graphs that can be handled by the TSE algorithm with minor modifications.

In a strongly connected process graph, all nodes have a path to a maximum ratio cycle, and all maximum ratio cycles have a path to the node s. This guarantees that m-values for all nodes eventually are determined repetitively using maximum ratio cycles and thus eventually repeat. For a non-strongly connected process graph the behavior of the m-value for a node v can be characterized as one of the following two:

1. $m(v_{(k)})$ eventually repeat with period $r(v)$ and occurrence period $\varepsilon^\star(v)$, i.e., $\forall k \geq k^\star(v)$:

$$m(v_{(k+\varepsilon^\star(v))}) - m(v_{(k)}) = r(v)\varepsilon^\star(v)\,.$$

   Note that $r(v)$ may be less than the maximum ratio r.

2. There is no path from $v_{(k)}$ to $s_{(\beta)}$, i.e., $\forall k : v_{(k)} \not\rightsquigarrow s_{(\beta)}$.

Depending on the behavior of the m-values, we have three classes of non-strongly connected process graphs:

1. For all nodes v, $v_{(k)} \rightsquigarrow s_{(\beta)}$ and furthermore all the m-values for all nodes repeat with the same periods, $r(v)$ and $\varepsilon^\star(v)$ (though not necessarily with r and $\varepsilon^\star$). The TSE algorithm can be used without modifications.

2. For all nodes v, if $v_{(k)} \rightsquigarrow s_{(\beta)}$ the m-values repeat with the same period (though not necessarily with the maximum ratio, r). However, some nodes may not have a path to $s_{(\beta)}$. We can make a simple modification to the TSE algorithm to handle this case. Recall that when $v_{(k)} \not\rightsquigarrow s_{(\beta)}$, we can assign an arbitrary constant to $m(v_{(k)})$, line 2 of the algorithm in Figure 4. In particular, we choose these m-values such that $r(v)$ and $\varepsilon^\star(v)$ are equal to the period of the events that have a path to $s_{(\beta)}$. This ensures that the matrix $\mathbf{S}$ is independent of the number of unfoldings and we can use the closure of $\mathbf{S}$ to analyze the infinite graph. A simple non-strongly connected process graph is shown in Figure 18.

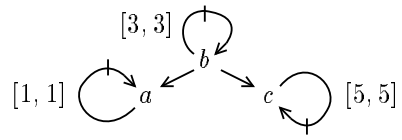

Figure 18: A non-strongly process graph. If $s = b$, nodes $a_{(k)}$ and $c_{(k)}$ do not have a path to $s_{(\beta)}$ and we can assign an arbitrary constants to $m(a_{(k)})$ and $m(c_{(k)})$. As $r(b) = 3$ and $\varepsilon^\star(b) = 1$, by chosing $m(a_{(k)})$ to $3 + m(a_{(k-1)})$ and similarly for $m(c_{(k)})$, all m-values repeat with the same period and the TSE algorithm can be applied for finding the maximum separation. Note that in this example the maximum ratio cycle is $c \overset{[5,5],1}{\longmapsto} c$ with the ratio $r = 5$.

3. There exist nodes whose m-values repeat with different ratios, i.e., $r(v)$ is different for different nodes. A simple example in this category is shown in Figure 19. This class of non-strongly connected process graphs can be handled by introducing a more complex functional algebra (the elements of the algebra are functions in two variables). We do not describe this algebra because it is not needed for the subsequent applications.

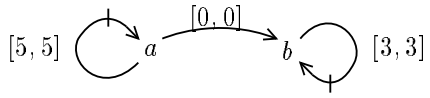

Figure 19: A non-strongly process graph where the m-values for different nodes repeat with different values. For $s = b$, we have $r(a) = 5$ and $r(b) = 3$.

4.8 Efficiency Considerations

There are two potential inefficiencies associated with the TSE algorithm. These inefficiencies are not usually encountered in practical applications.

1. Both $\varepsilon^\star$ and $k^\star$ depend on the delay ranges and are not polynomial in the size of the process graph.

2. The size of the representation of a particular function may be as large as the number of paths between the two events related by the function.

Point 1 is potentially serious, however in most realistic process graphs, $\varepsilon^\star = 1$ (see [6]). $k^\star$ is more of a concern because it can be large if there exists a cycle c such that $d(c)/\varepsilon(c)$ is almost equal to r.

Although the time required to perform the scalar operations is linear in the size of the operands, this does not imply that a polynomial number of scalar semiring operations can be performed in polynomial time. In fact, the size of the functions can potentially be as large as the number of paths between the vertices that the functions relate. In practice the functions can be efficiently pruned (see Section B.1) and the size of the functions seems to grow linearly with respect to the size of the process graph.

4.9 Examples

The details of the TSE algorithm are applied to some simple examples. The first example is from Figure 1 for $s = t = a$ and $\beta = 1$. The m-values repeat after three unfolding relative to the $s_{(\beta)}$ node ($k^\star = 3$) or $k^\star + \beta = 4$ unfoldings relative to the $t_{(0)}$ node (see Figure 15). The repetition period is 1, i.e., $\varepsilon^\star = 1$. Calling UNFOLD causes the construction of $G_{(i)}$ for $i = 1, 2, 3$, as illustrated in Figure 11. The bounds computed for these three graphs are:

i	$[\Delta^\perp, \Delta^\top]$
1	$[10, \infty]$
2	$[24, 25]$
3	$[25, 25]$

Thus after three unfoldings the bounds converge and the maximum separation of $\Delta = 25$ is reported. For this separation analysis it is not necessary to construct **R** and **S** and perform the closure, i.e., the TSE algorithm will stop at line 4. However, for the sake of illustration, we show the decomposition and the corresponding matrices constructed if the bounds had not converged in line 4.

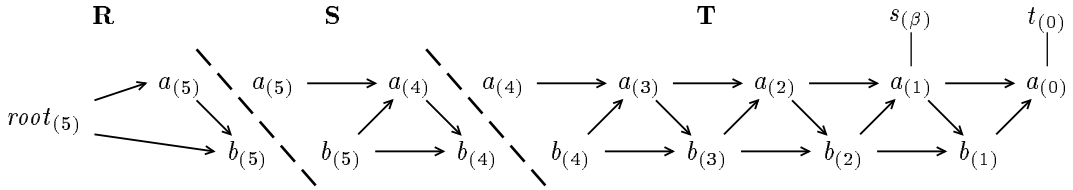

Figure 20: A decomposed unfolded process graph corresponding to the process graph in Figure 1.

Figure 20 shows the unfolded process graph. The segment represented by the matrix **T** is unfolded $k^\star + \beta = 4$ times. We chose the cutset $X_0 = \{a_{(4)}, b_{(4)}\}$ such that the m-values for all nodes left of X_0 repeat. Because $\varepsilon^\star = 1$, we have that $\mathbf{R} = \mathbf{R}_i$ and $\mathbf{S} = \mathbf{S}_i$ for $i \geq 0$. The matrices are

$$\mathbf{R} = \begin{pmatrix} \{\langle 0,0\rangle\} & \{\langle 1,0\rangle\} \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} \{\langle 5,0\rangle\} & \{\langle 6,0\rangle\} \\ \{\langle 2,0\rangle\} & \{\langle 15,0\rangle\} \end{pmatrix}$$
$$\mathbf{T} = \begin{pmatrix} \{\langle 46,25\rangle\} \\ \{\langle 55,25\rangle\} \end{pmatrix}$$

The closure of $\mathbf{S}$ is:

$$\mathbf{S}^* = \begin{pmatrix} \overline{1} \oplus \{\langle \infty,0\rangle\} & \{\langle \infty,0\rangle\} \\ \{\langle \infty,0\rangle\} & \overline{1} \oplus \{\langle \infty,0\rangle\} \end{pmatrix}$$

yielding the final product

$$F = \mathbf{R}\,\mathbf{S}^*\,\mathbf{T} = (\{\langle \infty,25\rangle\})\ .$$

The maximum separation between a_{k-1} and a_k for $k \geq 4$ is computed from the function $F = \{\langle \infty,25\rangle\}$, i.e., $\Delta_{\geq 4} = F(M(root_{(5)})) - m(a_{(0)}) = F(0) - 0$, yielding 25.

In Figure 13 we presented an example where the bounds on Δ do not converge. We now show the details of this example. We have $s = a$, $t = e$, and $\beta = 0$. We get $k^\star = 1$ and $\varepsilon^\star = 1$. Figure 21 shows a portion of the unfolded process graph. The functions at $b_{(0)}$ and $d_{(0)}$ are $\{\langle 0,0\rangle\}$ and $\{\langle 3,\infty\rangle\}$,

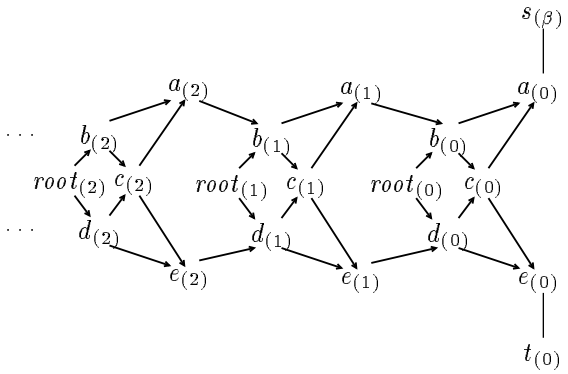

Figure 21: Portion of the unfolded process graph for the examples in Figure 13.

respectively. One unfolding back, the functions are $F_{b_{(1)} \mapsto e_{(0)}} = \{\langle 0,0\rangle\}$ and $F_{d_{(1)} \mapsto e_{(0)}} = \{\langle 3,3\rangle\}$. These functions will remain the same for all larger relative occurrence indices as $k^\star + \beta = 1$ and $\varepsilon^\star = 1$, i.e., for all $i \geq 1$:

$$F_{b_{(i)} \mapsto e_{(0)}} = \{\langle 0,0\rangle\}$$
$$F_{d_{(i)} \mapsto e_{(0)}} = \{\langle 3,3\rangle\}\ .$$

For all unfoldings k, $F_{root_{(k)} \mapsto e_{(0)}} = \{\langle 0,0\rangle\}$, resulting in $\Delta_k = F_{root_{(k)} \mapsto e_{(0)}}(0) - m(e_{(0)}) = 0 - 0$, i.e., $\Delta^\perp = 0$ for all $k \geq 1$. The upper bound is obtained from the cutset $X = \{b_{(k)}, d_{(k)}\}$. We get $\Delta^\top = \max(0,3) - 0 = 3$ for all $k \geq 1$. Thus, the bounds never converge.

However, by computing $\mathbf{R}$ and $\mathbf{S}$, we can analyze the infinite graph and realize that there is no way to achieve the bound at node d, thus 0 is the maximum separation between a_k and e_k, for all $k \geq 0$. Like in the previous examples, the matrices $\mathbf{R}$ and $\mathbf{S}$ are particularly simple because $\varepsilon^\star = 1$.

We get

$$\begin{aligned}
\mathbf{R} &= \begin{pmatrix} \{\langle 0,0\rangle\} & \{\langle -3,-3\rangle\} \end{pmatrix} \\
\mathbf{S} &= \begin{pmatrix} \{\langle 0,0\rangle\} & \{\langle -3,-3\rangle\} \\ \{\langle 3,0\rangle\} & \{\langle 0,0\rangle\} \end{pmatrix} \\
\mathbf{T} &= \begin{pmatrix} \{\langle 0,0\rangle\} \\ \{\langle 3,3\rangle\} \end{pmatrix}
\end{aligned}$$

The closure of $\mathbf{S}$ is:

$$\mathbf{S}^* = \begin{pmatrix} \overline{1} & \{\langle -3,-3\rangle\} \\ \{\langle 3,0\rangle\} & \overline{1} \end{pmatrix}$$

yielding the final product

$$F = \mathbf{R}\,\mathbf{S}^*\,\mathbf{T} = (\{\langle 0,0\rangle\})\ .$$

From $F(0) - 0 = 0$ we get $\Delta = \Delta_{\geq 1} = 0$.

Finally, we show the result of running the TSE algorithm on the example in Figure 14. Recall that this example requires the process graph to be unfolded 149 times before the bounds converge. However, after just 9 unfoldings the m-values repeat. Using the cutset $X = \{d_{(1)}, b_{(2)}, f_{(0)}\}$ and unfolding 11 times we get the bounds $\Delta^{\perp} = 68$ and $\Delta^{\top} = 188$. We get the following matrices ($\varepsilon^* = 1$):

$$\begin{aligned}
\mathbf{R} &= \begin{pmatrix} \{\langle 100,0\rangle\} & \{\langle -78,-78\rangle\} & \{\langle -219,-219\rangle\} \end{pmatrix} \\
\mathbf{S} &= \begin{pmatrix} \{\langle 100,0\rangle\} & \{\langle -89,-89\rangle\} & \overline{0} \\ \{\langle 0,0\rangle\} & \{\langle 1,0\rangle\} & \{\langle -140,-140\rangle\} \\ \overline{0} & \{\langle 0,0\rangle\} & \{\langle -1,-1\rangle\} \end{pmatrix} \\
\mathbf{T} &= \begin{pmatrix} \{\langle 58,58\rangle, \langle 157,57\rangle, \langle 256,56\rangle, \langle 355,55\rangle, \langle 454,54\rangle, \langle 553,53\rangle, \langle 652,52\rangle, \langle 751,51\rangle, \langle 951,50\rangle\} \\ \{\langle 132,131\rangle, \langle 134,122\rangle, \langle 136,113\rangle, \langle 138,104\rangle, \langle 140,95\rangle, \langle 142,86\rangle, \langle 144,77\rangle, \langle 146,68\rangle, \\ \langle 148,59\rangle, \langle 156,56\rangle, \langle 255,55\rangle, \langle 354,54\rangle, \langle 453,53\rangle, \langle 552,52\rangle, \langle 651,51\rangle, \langle 851,50\rangle\} \\ \{\langle 269,188\rangle, \langle 353,53\rangle, \langle 452,52\rangle, \langle 551,51\rangle, \langle 751,50\rangle\} \end{pmatrix}
\end{aligned}$$

The closure of $\mathbf{S}$ is:

$$\mathbf{S}^* = \begin{pmatrix} \overline{1} \oplus \{\langle \infty,0\rangle\} & \{\langle \infty,0\rangle\} & \{\langle \infty,-140\rangle\} \\ \{\langle \infty,0\rangle\} & \overline{1} \oplus \{\langle \infty,0\rangle\} & \{\langle \infty,-140\rangle\} \\ \{\langle \infty,0\rangle\} & \{\langle \infty,0\rangle\} & \overline{1} \oplus \{\langle \infty,-140\rangle\} \end{pmatrix}$$

yielding the final product

$$F = \mathbf{R}\,\mathbf{S}^*\,\mathbf{T} = (\{\langle \infty,131\rangle\})\ .$$

From $\Delta_{\geq 11} = F(0) - 0 = 131$ we get $\Delta = \max(131, 68) = 131$.

5 Applications

The TSE analysis is fundamental for performance analysis, timing verification, and optimization of concurrent systems. Classical performance measures can be derived based on the maximum

separation in time of events. Bounds on the latency between two events s and t are determined from a separation analysis of $\tau(t_k) - \tau(s_k)$. Bounds on the cycle period for event s can be obtained from $\tau(s_k) - \tau(s_{k-1})$. These measures give best and worst case delays from one event to the next. The best and worst n-term moving averages can be obtained by computing $\frac{\tau(s_{k+n})-\tau(s_k)}{n}$.

Timing verification is another area of application of the TSE analysis. Consider a timing constraint that specifies the maximum time, Δ, that may elapse from an event s to its response t. It must be verified that $\tau(t_k) - \tau(s_k) \leq \Delta$ for all k, which is directly obtainable from a TSE analysis. Similarly, a constraint specifying that at least a given amount of time, δ, must pass between two events, i.e., $\delta \leq \tau(t_k) - \tau(s_k)$ corresponds to the lower bound from the TSE analysis.

In this section, we present four specific examples that demonstrate the applicability and efficacy of the TSE algorithm.

5.1 Memory Management Unit

Consider an edge $u_{k-\varepsilon} \stackrel{[d,D]}{\longmapsto} v_k$ in an arbitrary process graph. If the minimum time separation between $u_{k-\varepsilon}$ and v_k is *larger* than D, event $u_{k-\varepsilon}$ will never constrain the time of event v_k, i.e., v_k must always wait for some other event to occur, and the edge from $u_{k-\varepsilon}$ can be removed from the process graph without changing the behavior of the system.

This idea can be used to remove redundant circuitry in asynchronous circuits given (conservative) bounds on the actual delays of a speed-independent design. Superfluous edges can be removed by analyzing the process graph corresponding to the circuit. This approach has been taken by Myers and Meng [25] who use an inexact timing analysis algorithm, i.e., the algorithm doesn't necessarily give tight bounds on separation times. Clearly, being able to obtain tight bounds potentially enables the removal of more edges.

One of the examples in [25] is a memory management unit (MMU) designed to interface to the Caltech Asynchronous Microprocessor [23]. The process graph (for one of the possible execution modes of the MMU) is shown in Figure 22.

For the chosen delay intervals, $k^\star = 1$ and $\varepsilon^\star = 1$. Analyzing the 23 edges using our exact algorithm takes 0.6 seconds on a SPARC 2. The analysis results in the removal of six edges from the process graph or equivalently, the removal of six transistors from the circuit. This is the same result as in [25].

5.2 Asynchronous Microprocessor

A subset of the Caltech Asynchronous Microprocessor [23] has been modeled and analyzed using the techniques described in this paper. A block diagram of the processor is shown in Figure 23.

The process graph for this simplified model consists of 60 events and 127 edges. Using our implementation of the techniques described in this paper, a separation analysis can be performed in under 2 seconds on a SPARC 2. For example, if all the delays associated with rules within the blocks of Figure 23 are in the range $[0, 1]$ (small in comparison to the delays associated with computation and data transfer) and we make the following assignments of delay ranges to the delay elements modeling the computations:

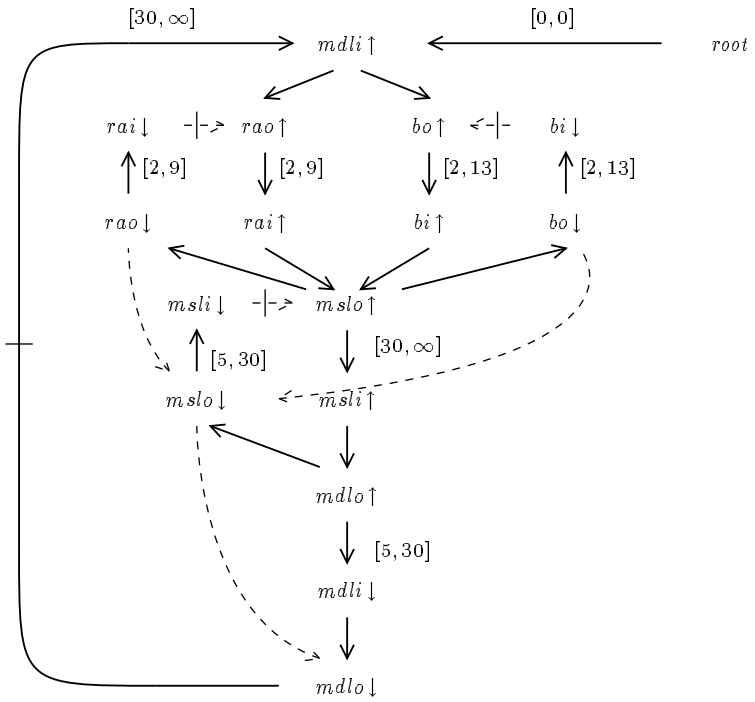

Figure 22: Process graph for memory management unit (from [25]). All unmarked edges have $[0, 1]$ as the delay range.

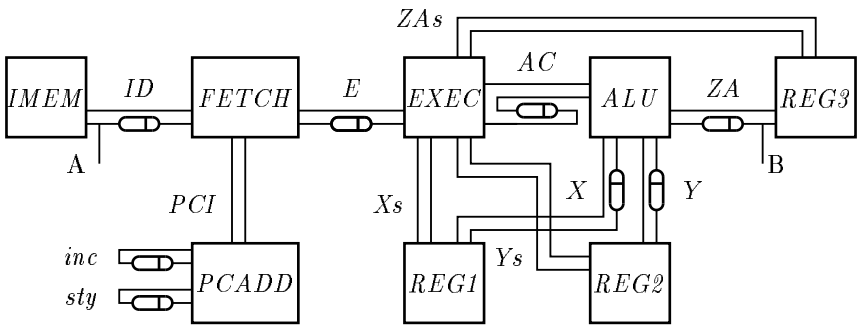

Figure 23: A block diagram of the Asynchronous Microprocessor. For a complete description of the process graph components within each block, see [6].

	Rise	Fall		Rise	Fall
ID	[50,51]	[30,31]	X	[30,31]	[30,31]
inc	[20,31]	[20,21]	Y	[30,31]	[30,31]
sty	[20,21]	[20,21]	AC	[20,21]	[20,21]
E	[20,21]	[20,21]	ZA	[40,51]	[40,41]

we obtain the minimum and maximum separations in cycle period seen at point A of 80 and 107. The minimum and maximum separation between identically numbered occurrences of rising transitions at point A and point B is 140 and 246. This last measurement corresponds to the pipeline latency of the microprocessor.

Computations of this type can be used to determine the real-time properties of the asynchronous microprocessor. This information is useful when interfacing the microprocessor to an external synchronous component, especially in cases where the synchronous component is clocked using a signal produced by the microprocessor.

5.3 STARI

STARI is a novel approach to high-bandwidth communication proposed by Greenstreet [13]. STARI combines synchronous and self-timed design techniques. The sender and receiver operate synchronously at the same clock rate, but the communication interface consists of an asynchronous FIFO queue. The overall structure is shown in Figure 24 (from [13, Figure 5]).

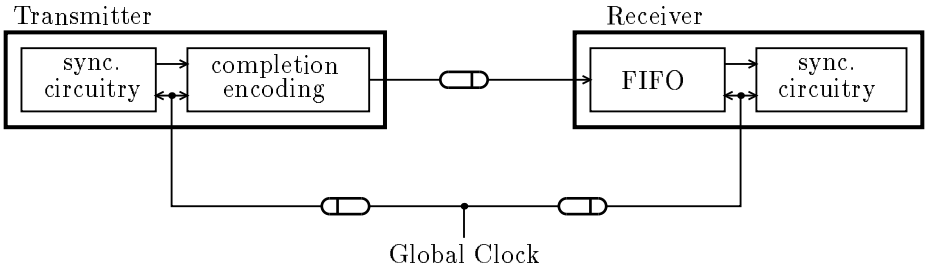

Figure 24: STARI communication.

The idea in STARI is to time the system such that the FIFO operates at the speed of the clock, accepting a new data item every clock cycle. The system is initialized to a state where the FIFO is half full. By taking the absolute delays (indicated by delay elements in Figure 24) into account in the initialization phase, arbitrary clock skews and transmission delays can be tolerated. The FIFO makes the system tolerant to dynamic changes of the delays, i.e., STARI is tolerant to variations in clock skew, to variations in the clock period and to variations in internal delays. If the variations becomes too large, the FIFO overflows or underflows and STARI fails. The amount of variation of the delays that can be tolerated depends on the length of the FIFO, the clock period, and the delay of the FIFO elements.

Greenstreet [13] has derived sufficient timing conditions under which STARI operates correctly. The correctness proof for these conditions is quite complicated (approx. 20 pages). Here we show how the maximum separation analysis can be used for timing verification of a STARI implementation. Given the clock period, the delay ranges for FIFO elements and the variation in clock skews, we can prove whether STARI is timed correctly. Consider a STARI implementation with a three stage FIFO (followed by a latch controlled by the receivers clock). The control structure is shown

in Figure 25. Figure 26 shows the corresponding process graph. The marks on the edges indicate

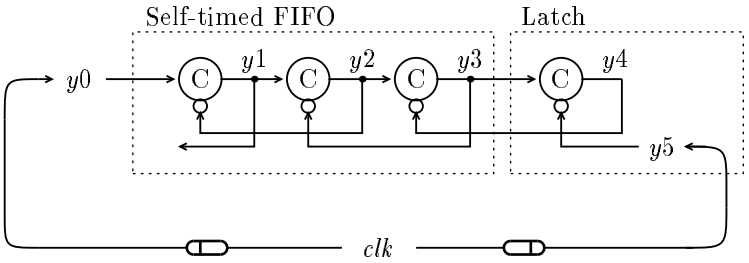

Figure 25: Control structure of STARI implementation with a three stage FIFO followed by a latch. The logic symbols denote Muller-C elements.

the initial state of a half full FIFO: the two first stages of the FIFO wait for a request from the sender while the third stage waits for an acknowledge by the receiver. The process graph is not strongly connected because the FIFO is not constraining the clock period.

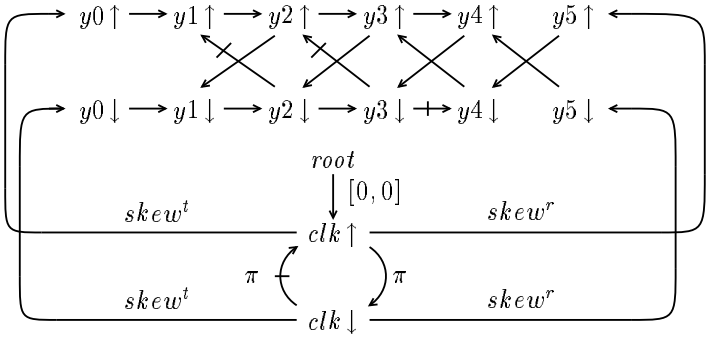

Figure 26: Process graph for STARI implementation.

For the correct operation of STARI, the synchronous components (i.e., the sender and the receiver) must adhere to the asynchronous protocol of the FIFO; the sender can only insert a new data item in the FIFO when the previous one has been acknowledged, and the receiver can only acknowledge a data item after it has been output from the FIFO. Referring to the signals in Figure 25, the requirements for correct operation of STARI are that a transition at $y0$ is acknowledge by $y1$ before another $y0$ transition, and similarly that $y4$ changes before it is acknowledged by $y5$. Thus, the following four timing relations need to hold for all possible executions in order to guarantee correct operation:

$$\tau(y1\uparrow_\alpha) \leq \tau(y0\downarrow_\alpha) \tag{18}$$

$$\tau(y1\downarrow_\alpha) \leq \tau(y0\uparrow_{\alpha+1}) \tag{19}$$

$$\tau(y4\uparrow_\alpha) \leq \tau(y5\uparrow_{\alpha+1}) \tag{20}$$

$$\tau(y4\downarrow_\alpha) \leq \tau(y5\downarrow_\alpha) \tag{21}$$

We can verify (18) by determining the smallest Δ such that for all $\alpha, \tau(y1\uparrow_\alpha) - \tau(y0\downarrow_\alpha) \leq \Delta$. If $\Delta \leq 0$, (18) holds. The other three inequalities have similar analyses. Thus, by applying our algorithm four times we can verify the correct operation of the STARI protocol for all possible delay variations in the specified ranges.

Table 1 shows the result for different values for the clock period (π), and for different variations of clock skew to the transmitter ($skew^t$) and to the receiver ($skew^r$). The CPU time to verify the four conditions is less than a second on a SPARC 2.

π_d	$skew^t$	$skew^r$	$\Delta_{(18)}$	$\Delta_{(19)}$	$\Delta_{(20)}$	$\Delta_{(21)}$	OK
6	$[0,0]$	$[0,0]$	-3	-3	0	0	$\surd$
6	$[0,0]$	$[0,3]$	-3	-3	0	0	$\surd$
6	$[0,3]$	$[0,0]$	0	0	3	3	
6	$[0,0]$	$[0,6]$	0	0	3	3	
6	$[0,0]$	$[6,6]$	0	0	-3	-3	$\surd$
8	$[0,0]$	$[0,0]$	-5	-5	-4	-4	$\surd$
8	$[0,3]$	$[0,5]$	-2	-2	0	0	$\surd$
8	$[0,5]$	$[0,3]$	0	0	1	1	
8	$[0,0]$	$[0,12]$	0	0	7	7	
8	$[0,0]$	$[12,12]$	0	0	-5	-5	$\surd$
8	$[12,12]$	$[0,0]$	-5	-5	8	8	

Table 1: Timing analysis of STARI. π_d is the lower bound on the clock period (the upper bound is irrelevant for the correctness of STARI), $skew^t$ and $skew^r$ are the variations in skew to the transmitter and receiver, respectively. The delay through a C-element is set to $[2,3]$ in all cases. $\Delta_{(i)}$ is the maximum separation corresponding to equation (i). Correct operation of STARI is indicated with a checkmark in the OK column.

5.4 Isochronic Forks

An exact solution to the maximum separation problem can be used to determine whether or not an asynchronous circuit designed under the assumptions of the quasi–delay-insensitive model [22] will work correctly even if the *isochronic fork assumption* is relaxed. The isochronic fork assumption states that certain signals in the circuit that fan out to separate circuit elements arrive at their respective elements at the *same* time. This assumption is very strong; what is actually important is that the circuit behaves as if these signals are isochronic. However, we can check for correct behavior by performing various maximum separation analyses on a process graph that includes the timing information corresponding to both the operators and wires of the circuit.

As an example, the D-element (Figure 27) is a simple asynchronous component that sequences two four-phase handshakes. A process graph corresponding to the D-element and its environment is shown in left half of Figure 28. Assume now that the three forks of the system are no longer considered to be isochronic. This can be modeled by introducing new signals in the circuit corresponding to the ends of the forks and then adding new events to the process graph corresponding to the transitions on these new signals. The non-isochronicity of a fork is modeled as bounded delay intervals on the rules coming into the new events. The new process graph is shown in the right half of Figure 28 and is no longer strongly connected.

By performing maximum separation analyses on this graph, we can determine whether or not the violation of the isochronic fork assumption can cause the circuit to fail. In particular, we must

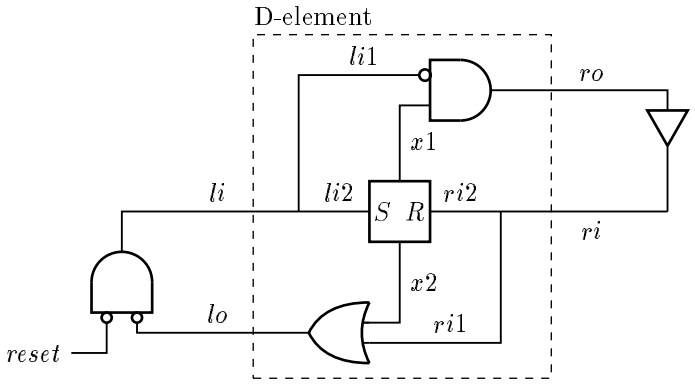

Figure 27: Circuit for the D-element connected to a trivial environment. The center element is an S-R latch.

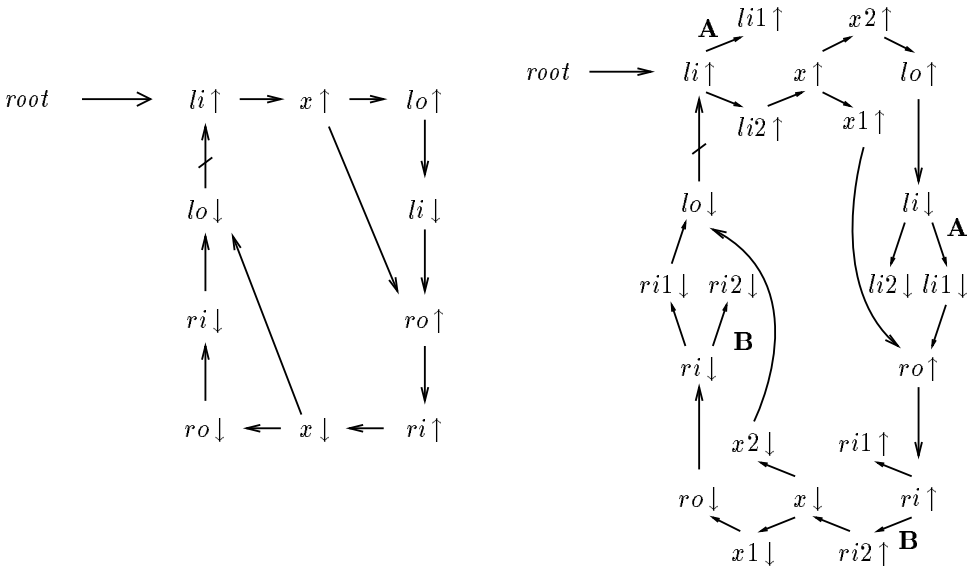

Figure 28: The left half shows the process graph for the D-element assuming isochronic forks. The right half shows the process graph for the same circuit if isochronic forks are not assumed.

show that

$$\tau(li1\uparrow_\alpha) - \tau(x1\uparrow_\alpha) \leq 0 \quad (22)$$
$$\tau(ri1\uparrow_\alpha) - \tau(x2\downarrow_\alpha) \leq 0 \quad (23)$$
$$\tau(li2\downarrow_\alpha) - \tau(ri2\uparrow_\alpha) \leq 0 \quad (24)$$
$$\tau(ri2\downarrow_{\alpha-1}) - \tau(li2\uparrow_\alpha) \leq 0 \quad (25)$$

for all α. By satisfying these inequalities, we know that the events in the process graph that have no fanout (i.e., the transitions of the signals in the circuit that are not acknowledged) have actually occurred before the outcome of the event is needed later in the execution of the circuit. For example, in order to insure that $ro\uparrow$ does not fire prematurely, we must know that $li1\uparrow$ has occurred before $x1\uparrow$, corresponding to (22). A similar argument holds for (23). (24) and (25) are needed to insure that the set and reset signals to the latch are not active simultaneously.

If we assume that all gate delays are in the range $[2,3]$ and all the wire delays are in the range $[0,1]$ except those specified explicitly in the table, we get the results shown in Table 2. The CPU time needed to verify the four conditions is less than half a second on a SPARC 2.

Arc **A**	Arc **B**	$\Delta_{(22)}$	$\Delta_{(23)}$	$\Delta_{(24)}$	$\Delta_{(25)}$	OK
$[0,1]$	$[0,1]$	-1	-1	-3	-3	$\surd$
$[0,2]$	$[0,1]$	0	-1	-3	-3	$\surd$
$[0,3]$	$[0,1]$	1	-1	-3	-3	
$[0,5]$	$[0,1]$	3	-1	-3	-3	
$[0,1]$	$[0,2]$	-1	-1	-3	-2	$\surd$
$[0,1]$	$[0,3]$	-1	-1	-3	-1	$\surd$
$[0,1]$	$[0,5]$	-1	-1	-3	1	

Table 2: Results of the maximum separation analysis for the D-element. $\Delta_{(i)}$ is the maximum separation corresponding to equation (i).

6 Conclusion

We have presented an efficient exact solution to a fundamental problem for timing analysis and verification of concurrent systems, namely, the determination of bounds on the separation in time between two arbitrary events. The major contribution of this paper is the structural decomposition of the infinitely unfolded process graph which allows the infinite graph to be implicitly analyzed to obtain the tightest possible bounds. This aspect of the solution and its algebraic formulation enables the algorithm to be efficient in practice. Furthermore, the algorithm handles a wide range of process graphs and is thus useful in a variety of domains.

The utility of the time separation of events algorithm for solving a wide range of practical problems has been demonstarted. These include: determination of execution times in large concurrent systems such as an asynchronous microprocessor, verification of a high-performance communication protocol, and verification of isochronous fork assumptions.

We are looking into adaptations of this technique to graphs that include conditional behavior and thus process an ever-larger class of graphs. This may require the exploration of tradeoffs between the tightness of the bounds and computation time that has not been a concern up to now because of the high efficiency of the algorithm in practice. In concert with this effort, we are also investigating other problem domains such as high-level synthesis and hardware/software co-design as potential application areas.

Acknowledgments

This work was supported by an NSF PYI Award (MIP-8858782), an NSF YI Award (MIP-9257987), by the DARPA/CSTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041), by an IBM Graduate Fellowship, and by the Technical University of Denmark.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] 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*, Lecture Notes in Computer Science #600, pages 28–73. Springer-Verlag, 1991.

[3] T. Amon and G. Borriello. An approach to symbolic timing verification. In *29th ACM/IEEE Design Automation Conference*, June 1992.

[4] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, 1992.

[5] Gaetano Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. Ph.D. thesis, University of California at Berkeley, 1988.

[6] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. Ph.D. thesis, California Institute of Technology, 1991. CS-TR-91-1.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sosta. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] G. Cohen, P. Moller, J. P. Quadrat, and M. Viot. Evaluation of discrete event systems. *Proceedings of the IEEE*, 77(1):39–58, January 1989.

[9] R. A. Cunighame-Green. *Minimax Algebra*. Number 166 in Lecture Notes in Economics and Methematical Systems. Springer-Verlag, 1979.

[10] S. Gaubert. *Théorie des Systèmes Linéaires dans les Dioïdes*. Ph.D. thesis, L'école Nationale Supérieure des Mines De Paris, 1993. In French.

[11] S. Gaubert and C. Klimann. Rational computation in dioid algebra and its application to performance evaluation of discrete event systems. In *Algebraic computing in control*, Lecture Notes in Computer Science # 165. Springer Velag, 1991.

[12] Mike Gordon. HOL: A proof generating system for higher-order logic. In G. Milne and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[13] M. R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. Ph.D. thesis, Princeton University, January 1993.

[14] I. N. Herstein. *Topics in algebra*. Blaisdell Publishing Company, 1964.

[15] F. Jahanian. Verifying properties of systems with variable timing constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 319–328, 1989.

[16] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[17] F. Jahanian and A. K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, 36:961–975, August 1987.

[18] F. Jahanian and D. A. Stuart. A method for verifying properties of modechart specifications. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 12–21, December 1988.

[19] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[20] A. R. Martello and S. P. Levitan. Temporal analysis of time bounded digital systems. In *Correct hardware design and verification methods : IFIP WG10.2 Advanced Research Working Conference, CHARME '93*, May 1993.

[21] A. R. Martello, S. P. Levitan, and D. M. Chiarulli. Timing verification using HDTV. In *27th ACM/IEEE Design Automation Conference*, pages 118–123, 1990.

[22] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, Reading, MA, 1990.

[23] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C.L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, MA, 1989. MIT Press.

[24] K. 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.

[25] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):106–119, June 1993.

[26] X. Nicolin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, September 1992.

[27] J. S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[28] E. S. Selmer. On the linear diophantine problem of Frobenius. *J.-Reine-Angew.-Math.*, 293/294:1–17, 1977.

[29] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms*. John Wiley and Sons, 1992.

[30] P. Vanbekbergen, G. Goossens, and H. De Man. Specification and analysis of timing constraints in signal transition graphs. In *European Design Automation Conference*, March 1992.

A Proof of Theorem 1

Let $\Delta(v_k)$ denote the maximum separation from node $s_{\alpha-\beta}$ to node v_k in the finite unfolded process graph $G_\alpha = \langle E_\alpha, R_\alpha \rangle$. We want to prove that $\Delta(v_k) = M(v_k) - m(v_k)$.

We begin by asserting that there does not exist an execution with a separation greater than $M(v_k) - m(v_k)$. Let τ be a timing assignment that achieves the maximum separation, i.e., $\tau(v_k) - \tau(s_{\alpha-\beta}) = \Delta(v_k)$. For convenience, we use $\max_r\{\cdot\}$ to denote

$$\max\left\{ \cdot \;\middle|\; r = u_j \overset{[d,D]}{\longmapsto} v_k \in R_\alpha \right\} .$$

For the case $v_k \rightsquigarrow s_{\alpha-\beta}$, we have from the definition of $M(v_k)$

$$\begin{aligned} M(v_k) - m(v_k) &= \max_r \{\min(0, M(u_j) - m(u_j) + D + m(v_k))\} - m(v_k) \\ &= \max_r \{\min(-m(v_k), M(u_j) - m(u_j) + D)\} \\ &= \min\left(-m(v_k), \max_r \{M(u_j) - m(u_j) + D\}\right) . \end{aligned} \tag{26}$$

We prove $\Delta(v_k) \leq M(v_k) - m(v_k)$ by proving $\Delta(v_k)$ to be less then each of the two parts of the minimization in (26). By definition of the m-values we have that any consistent timing assignment satisfies $\tau(s_{\alpha-\beta}) - \tau(v_k) \geq m(v_k)$, so $-\Delta(v_k) \geq m(v_k)$. Multiplying by -1 yields $\Delta(v_k) \leq -m(v_k)$.

The second part, $\Delta(v_k) \leq \max_r\{M(u_j) - m(u_j) + D\}$ is proven by topological induction. Basis is $v_k = \mathit{root}$, which holds trivially as *root* has indegree of zero. Now assume $\Delta(u_j) \leq M(u_j) - m(u_j)$. From the definition of a consistent timing assignment (1), we have $\tau(v_k) \leq \max_r \{\tau(u_j) + D\}$. Subtracting $\tau(s_{\alpha-\beta})$ on both sides yields:

$$\begin{aligned} \Delta(v_k) = \tau(v_k) - \tau(s_{\alpha-\beta}) &\leq \max_r \{\tau(u_j) - \tau(s_{\alpha-\beta}) + D\} \\ &\leq \max_r \{\Delta(u_j) + D\} \\ &\leq \max_r \{M(u_j) - m(u_j) + D\} . \end{aligned}$$

If $v_k \not\rightsquigarrow s_{\alpha-\beta}$, the minimization in (26) is omitted and the proof reduces to the second part above.

We now prove that there exist an execution where $M(v_k) - m(v_k)$ is the maximum separation, i.e., there exists a consistent timing assignment τ such that

$$\Delta(v_k) = \tau(v_k) - \tau(s_{\alpha-\beta}) = M(v_k) - m(v_k) . \tag{27}$$

Choose as the timing assignment:

$$\tau(v_k) = M(v_k) - m(v_k) + \tau(s_{\alpha-\beta}) . \tag{28}$$

This timing assignment trivially satisfies (27). It remains to be proved that τ is a legal execution, i.e., that τ satisfies (1) for all nodes. If $v_k \rightsquigarrow s_{\alpha-\beta}$, we substitute the expression (26) for $M(v_k) - m(v_k)$ into (28)

$$\begin{aligned} \tau(v_k) &= \min\left(-m(v_k), \max_r \{M(u_j) - m(u_j) + D\}\right) + \tau(s_{\alpha-\beta}) \\ &= \min\left(\tau(s_{\alpha-\beta}) - m(v_k), \max_r \{\tau(u_j) + D\}\right) . \end{aligned}$$

By the definition of min, $\tau(v_k) \leq \max_r\{\tau(u_j) + D\}$. We now prove

$$\max_r \{\tau(u_j) + d\} \leq \tau(v_k)\,,$$

by proving

$$\max_r \{\tau(u_j) + d\} \leq \max_r \{\tau(u_j) + D\}\,, \tag{29}$$

and

$$\max_r \{\tau(u_j) + d\} \leq \tau(s_{\alpha-\beta}) - m(v_k)\,.$$

The first follows from the fact that $d \leq D$ for all edges, and the second from

$$\begin{aligned} \max_r \{M(u_j) - m(u_j) + \tau(s_{\alpha-\beta}) + d\} &\leq \tau(s_{\alpha-\beta}) - m(v_k) \\ \max_r \{M(u_j) - m(u_j) + d\} &\leq -m(v_k)\,. \end{aligned} \tag{30}$$

From the definition of m-values, $m(u_j) \geq m(v_k) + d$ for all edges $u_j \overset{[d,D]}{\longmapsto} v_k \in R$ and thus

$$\max_r \{-m(u_j) + d\} \leq -m(v_k)\,. \tag{31}$$

As for all nodes u_j such that $u_j \rightsquigarrow s_{\alpha-\beta}$, $M(u_j) \leq 0$ (follows from the definition of the M-values), we have that (31) implies (30), concluding the proof. If $v_k \not\rightsquigarrow s_{\alpha-\beta}$ the timing assignment only need to satisfy (29), which follows from a equivalent argument.

So $\tau(v_k) - \tau(s_{\alpha-\beta}) \leq M(v_k) - m(v_k)$ for every timing assignment τ and there exists a timing assignment with $\tau(v_k) - \tau(s_{\alpha-\beta}) = M(v_k) - m(v_k)$, and thus $\Delta(v_k) = M(v_k) - m(v_k)$. One such timing assignment, having $\tau(\mathit{root}) = 0$, is:

$$\tau(v_k) = \begin{cases} 0 & \text{if } v_k = \mathit{root} \\ \min\left(m(\mathit{root}) - m(v_k), \max\left\{\tau(u_j) + D \;\middle|\; u_j \overset{[d,D]}{\longmapsto} v_k \in R\right\}\right) & \text{if } v_k \rightsquigarrow s_{\alpha-\beta} \\ \max\left\{\tau(u_j) + D \;\middle|\; u_j \overset{[d,D]}{\longmapsto} v_k \in R\right\} & \text{if } v_k \not\rightsquigarrow s_{\alpha-\beta} \end{cases}$$

B Data Structures and Algorithms

This section describes details necessary to implement the TSE algorithm efficiently. These include algorithms for performing scalar function operations, for finding an optimal cutset, and for computing the m-values.

B.1 Scalar Function Operations

Recall that a function $f \in \mathcal{F}$ is represented as a set of pairs

$$f = \{\langle l_1, w_1\rangle, \langle l_2, w_2\rangle, \ldots, \langle l_n, w_n\rangle\} \tag{32}$$

corresponding to the function

$$f(x) = \max\{\min(x + l_i, w_i) \mid 1 \leq i \leq n\}\,.$$

The following observation leads to an important efficiency optimization: If $l_i \geq l_j$ and $w_i \geq w_j$ for two pairs $p_i = \langle l_i, w_i \rangle$ and $p_j = \langle l_j, w_j \rangle$, then p_i subsumes p_j since for all x, $\min(x + l_i, w_i) \geq \min(x + l_j, w_j)$. Thus, a function (32) can always be represented as a list of pairs such that

$$l_1 < l_2 < \ldots < l_n \text{ and } w_1 > w_2 > \ldots > w_n \,. \tag{33}$$

Performing the maximum of two functions, f_1 and f_2, can clearly be done in time $|f_1| + |f_2|$. More surprisingly is it that the same holds when composing two functions. This follows from the following theorem which leads directly to an algorithm composing two functions in linear time.

Theorem 3

$|f_1 \otimes f_2| < |f_1| + |f_2|$.

Proof: Proof by induction on $|f_1| + |f_2|$.

Basis: $|f_1| = |f_2| = 1$. We have $|f_1 \otimes f_2| = 1$ which is less than $|f_1| + |f_2| = 2$.

Inductive step: Let $f_1 = \{\langle l_1, w_1 \rangle, \langle l_2, w_2 \rangle, \ldots, \langle l_n, w_n \rangle\}$ and $f_2 = \{\langle L_1, W_1 \rangle, \langle L_2, W_2 \rangle, \ldots, \langle L_m, W_m \rangle\}$ be two functions on the form (33). Let $p_{n,m} \in f_1 \otimes f_2$ be

$$p_{n,m} = \begin{cases} \langle \infty, f_2(w_n) \rangle & \text{if } l_n = \infty \\ \langle l_n, w_n \rangle \otimes \langle L_m, W_m \rangle & \text{otherwise} \end{cases} \tag{34}$$

It straightforward to prove that $f_1 \otimes f_2 = f_1' \otimes f_2' \oplus p_{n,m}$, where either $f_1' = f_1 - \{\langle l_n, w_n \rangle\}$ and $f_2' = f_2$ or $f_1' = f_1$ and $f_2' = f_2 - \{\langle L_m, W_m \rangle\}$. In both cases we have from the inductive hypothesis that $|f_1' \otimes f_2'| < |f_1'| + |f_2'| = n + m - 1$ and thus $|f_1 \otimes f_2| = |f_1' \otimes f_2'| + 1 < n + m - 1 + 1 = n + m$.

□

Let f be a function satisfying (33). The scalar closure operation of f,

$$f^* = \overline{1} \oplus f \oplus f^2 \oplus f^3 \oplus \cdots ,$$

can be efficiently computed by:

$$f^* = \begin{cases} \overline{1} \oplus \{\langle \infty, w_q \rangle\} & \text{if } l_n > 0 \\ \overline{1} & \text{if } l_n \leq 0 \end{cases} \tag{35}$$

where w_q is the w-component of the first pair with positive l, i.e., $l_q > 0$ and if $q > 1$ then $l_{q-1} \leq 0$. Closing a function can clearly be done in linear time.

Function maximization and composition is used when constructing the functions for the **R**, **S**, and **T** matrices. The closure of a function is needed when forming the closure of **S**. A matrix closure algorithm [1] can be used to compute $\mathbf{S}^*$ because $(\mathcal{F}, \oplus, \otimes, \overline{0}, \overline{1})$ forms a closed semiring. The closure of an $n \times n$ matrix can be performed in $O(n^3)$ scalar function operations ($n = O(E')$), see Figure 29.

```
MATRIX-CLOSURE(S)
1  n ← rows[S]
2  D^(0) ← S ⊕ I
3  for k ← 1 to n do
4       for i ← 1 to n do
5            for j ← 1 to n do
6                 d_ij^(k) ← d_ij^(k-1) ⊕ d_ik^(k-1) ⊗ (d_kk^(k-1))* ⊗ d_kj^(k-1)
7  return D^(n)
```

Figure 29: Algorithm for forming the closure of a matrix **S**.

B.2 Computing m-values

We have identified two approaches for computing m-values. One is to determine the maximum ratio cycles in the process graph and use these to derive $\varepsilon^\star$ and $k^\star$. The other approach is to represent the m-value computation using a $(\max, +)$-algebra, and find m-values for all occurrences by closing a matrix in this algebra. Which one is appropriate depends on the use of the TSE algorithm. If many separation analysis are done on the same process graph (with different $s_{(\beta)}$ events) or non-strongly connected process graphs are analyzed, the algebraic approach is most efficient.

A primal-dual algorithm [6] can be used to efficiently determine the maximum ratio r and the maximum ratio cycles. Computing $\varepsilon^\star$ is complicated by the fact that there may be multiple maximum ratio cycles; m-values computed for different events may use different maximum ratio cycles.

Let G^* denote the sub-graph of G' including only edges on maximum ratio cycles and let G_i^* denote the i^{th} strongly connected component of G^*. If an event, v, can get its m-value from two different maximum ratio cycles, c_1 and c_2, and these two cycles are in different strongly connected components, then the occurrence period of v, $\varepsilon^\star(v)$, is

$$\varepsilon^\star(v) = \text{lcm}(\varepsilon(c_1), \varepsilon(c_2))\,,$$

where $\varepsilon(c)$ is sum of the ε-values of the edges on the cycle c. On the other hand, if c_1 and c_2 are in the same strongly connected component we get

$$\varepsilon^\star(v) = \text{gcd}(\varepsilon(c_1), \varepsilon(c_2))\,.$$

An approach to computing $\varepsilon^\star$ is to enumerate all cycles in G_i^* and compute $\varepsilon^\star$ as

$$\varepsilon^\star = \text{lcm}\{\text{gcd}\{\varepsilon(c) \mid c \in G_i^*\}\}\,.$$

The problem with this approach is that there may be an exponential number of maximum ratio cycles, making it potentially expensive to enumerate all of them. Instead we can compute an upper bound on $\varepsilon^\star$ by using the smallest ε value for each strongly connected component:

$$\varepsilon^\star \leq \text{lcm}\,\{\min\{\varepsilon(c) \mid c \in G_i^*\}\}\ .$$

This is an upper bound because $\gcd(a,b) \leq \min(a,b)$ for any two numbers a and b. Finding the cycle in G_i^* with minimal ε-sum can be done efficiently using an all pairs shortest path algorithm.

The other important number related to the m-values is $k^\star$, i.e., the number of unfoldings before all m-values repeat. Let $k^\star(v)$ denote the number of unfoldings before $m(v_{(k)})$ repeats (with occurrence period $\varepsilon^\star(v)$). We have that

$$k^\star = \max\left\{k^\star(v) \mid v \in E'\right\} .$$

We now consider the problem of finding $k^\star(v)$ for a given node $v \in E'$. For a strongly connected process graph we know that all nodes eventually get their m-values from maximum ratio cycles. As we unfold the process graph we can detect when $m(v_{(k)})$ is obtained from a maximum ratio cycle. If the maximum ratio cycle has $\varepsilon(c) = 1$, all subsequent $m(v_{(k)})$ values can be obtained repeatingly using the cycle.

However, if the maximum ratio cycles have ε-values larger than one, further unfoldings may be necessary before the m-values repeat. This is best illustrated using a simple example. Assume node $v_{(k_0)}$ can get its m-value from two maximum ratio cycles, c_1 and c_2, with $\varepsilon(c_1) = 3$ and $\varepsilon(c_2) = 5$. Also, let c_1 and c_2 be in the same strongly connected component, then $\varepsilon^\star(v) = \gcd(3,5) = 1$. Although $v_{(k_0)}$ gets the m-value from maximum ratio cycles, this does not guarantee that the m-values repeat yet. We can certainly compute $m(v_{(k_0+\varepsilon(c_1))})$ from a maximum ratio cycle by traversing c_1 one additional time. In fact, all subsequent m-values for nodes separated by $\varepsilon(c_1)$ in occurrence index get the m-value from a maximum ratio cycles. Similarly can $m(v_{(k_0+n\varepsilon(c_2))})$ (for $n \geq 0$) be obtained from the cycle c_2. In fact, every $m(v_{(k_0+n\varepsilon(c_1)+m\varepsilon(c_2))})$ for $n, m \geq 0$ can be obtained from maximum ratio cycles. Exactly when $m(v_{(k_0+n)})$ for all $n \geq 0$ is obtained from maximum ratio cycles is determined as the solution to the Frobenius problem [28] given $\varepsilon(c)$ for the maximum ratio cycles c.

For two variables, a and b, the Frobenius problem can be solved exact, and the solution is $(a-1)(b-1)$. That is, for the example, we need $(\varepsilon(c_1)-1)(\varepsilon(c_2)-1) = 8$ further unfoldings before $m(v_{(k)})$ repeats. More generally, we have

$$k^\star(v) = k_0 + \text{frobenius}\left\{\varepsilon(c) \mid c \in G_i^*\right\} .$$

For more than two variables, no exact solution to the Frobenius problem is known, but several bounds exists [28]. Note that any subset of $\varepsilon^\star$-values also gives a bound on $k^\star(v)$, and if there exists a maximum ratio cycle c with $\varepsilon(c) = 1$, the solution to the Frobenius problem is 0.

The m-values can also be computed using a *dioid* algebra [8, 11, 10]. The lower delay bound and the occurrence index offset for all edges in the process graph are represented as elements of the dioid algebra in an $|E'| \times |E'|$ matrix. By closing this matrix (in the dioid algebra) the m-values for all nodes are computed. An entry of the closure of this matrix contains functions that, given an occurrence index, returns the m-value. In $O(|E'|^3)$ scalar dioid operations the m-values at all event occurrences for every $s \in E'$ are computed.

This approach works regardless of the structure of the process graph, and the entries in the closed dioid matrix can be used to determine everything about the behavior of the m-values, including $\varepsilon^\star$ and $k^\star$. This is very useful when analyzing non-strongly connected process graphs, see Section 4.7.

B.3 Cutsets

Determining a small cutset of the unfolded process graph can greatly improve the practical performance of the TSE algorithm. The dimension of the **S** matrix is determined by the dimension of

the cutset. Closing $\mathbf{S}$ takes $O(n^3)$ where n is the dimension of the cutset. Clearly reducing n can drastically improve the practical execution time of the TSE algorithm.

It is simple to find a cutset X for an unfolded process graph:

$$X = \left\{ v_{(i)} \;\middle|\; u \overset{[d,D],\varepsilon}{\longmapsto} v \in E', 0 \leq i < \varepsilon \right\}. \tag{36}$$

However, this is not necessarily the smallest cutset. We can use a max-flow algorithm to determine a smaller cutset. From the max-flow min-cut theorem [29] it follows that for a flow network (with source s and sink t) we can find a minimal set of edges whose removal disconnects all path from s to t. We can find a minimal cutset (i.e., a minimal set of *vertices* whose removal disconnects the graph) for a finite unfolded process graph G_k from a flow network G_k^f. The flow network is constructed by splitting all nodes (except the source and sink) into two nodes with an edge between them. All edges are assigned capacity 1. Finding the minimal edge cut of G_k^f corresponds to a minimal vertex cut of G_k (Menger's theorem, [29]).

In this context, the Ford-Fulkerson max-flow algorithm is efficient. The complexity of Ford-Fulkerson is $O(EU)$, where E is the number of edges in the flow network and U is the maximum flow. We can bound U as a cutset will always be smaller than $\varepsilon_{max}|E'|$, where $\varepsilon_{max} = \max\{\varepsilon \mid u \overset{[d,D],\varepsilon}{\longmapsto} v \in E'\}$. I.e., the maximum flow is bounded from above by $\varepsilon_{max}|E'|$, therefore the complexity of the Ford-Fulkerson algorithm is $O(N|E'||R'|)$, where $N \geq \varepsilon_{max}$ is the number of times the process graph is unfolded.

A cutset for the process graph in Figure 14 computed using (36) is $\{a_{(0)}, b_{(0)}, c_{(0)}, d_{(0)}, e_{(0)}, f_{(0)}\}$, while a minimal cutset is $\{b_{(2)}, d_{(1)}, f_{(0)}\}$.