A Formal Model For Real-Time Parallel Computation

Peter Hui Satish Chikkagoudar

Pacific Northwest National Laboratory
Washington, USA

peter.hui@pnnl.gov satish.chikkagoudar@pnnl.gov

The imposition of real-time constraints on a parallel computing environment— specifically high-performance, cluster-computing systems— introduces a variety of challenges with respect to the formal verification of the system's timing properties. In this paper, we briefly motivate the need for such a system, and we introduce an automaton-based method for performing such formal verification. We define the concept of a consistent parallel timing system: a hybrid system consisting of a set of timed automata (specifically, timed Büchi automata as well as a timed variant of standard finite automata), intended to model the timing properties of a well-behaved real-time parallel system. Finally, we give a brief case study to demonstrate the concepts in the paper: a parallel matrix multiplication kernel which operates within provable upper time bounds. We give the algorithm used, a corresponding consistent parallel timing system, and empirical results showing that the system operates under the specified timing constraints.

1 Introduction

Real-time computing has traditionally been considered largely in the context of single-processor and embedded systems, and indeed, the terms real-time computing, embedded systems, and control systems are often mentioned in closely related contexts. However, real-time computing in the context of multinode systems, specifically high-performance, cluster-computing systems, remains relatively unexplored. It can be argued that one reason for the relative dearth of work in this area is the lack of scenarios to date which would require such a system. Previously [11, 12], we have motivated the emerging need for such an infrastructure, giving a specific scenario related to the next generation North American electrical grid. In that work, we described the changes and challenges in the power grid driving the need for much higher levels of *computational resources* for power grid operations. To briefly summarize (and to provide some motivational context for the current work), many of these computations— particularly floating-point intensive simulations and optimization calculations ([2, 3, 8, 9, 10])—can be more effectively done in a centralized manner, and the amount and scale of such data is estimated by some [11, 12] to be on the order of terabytes per day of streaming sensor data (e.g. Phasor Measurement Units (PMUs)), with the need to analyze the data within a strict cyclical window (every 30ms), presumably with the aid of high-performance, parallel computing infrastructures. With this in mind, the current work is part of a larger research effort at Pacific Northwest National Laboratory aimed at developing the necessary infrastructure to support an HPC cluster environment capable of processing vast amounts of streaming sensor data under hard real-time constraints.

While verifying the timing properties of a more traditional (e.g. embedded) real-time system poses complex questions in its own right, imposing real-time constraints on a parallel (cluster) computing environment introduces an entirely new set of challenges not seen in these more traditional environments. For example, in addition to standard real-time concepts such as *worst-case execution time* (WCET), real-time parallel computation introduces the necessity of considering *worst case transmission time* when

C. Artho and P.C. Ölveczky (Eds.): FTSCS 2012
EPTCS 105, 2012, pp. 39–55, doi:10.4204/EPTCS.105.4

communicating over the network between nodes, as well as the need to ensure that timing properties of one process do not invalidate those of the entire parallel process as a whole.

These are but two examples of the many questions which must be addressed in a real-time parallel computing system; certainly there are many more questions than can be addressed in a single paper. To this end, we introduce a simple, event driven, automata-based model of computation intended to model the timing properties of a specific class of parallel programs. Namely, we consider SPMD (Single Program, Multiple Data), parent-child type programs, in part because in practice, many parallel programs—including many prototypical MPI-based [13, 15] programs— fall into this category. We give an example of such a program in Section 3. This model is typified by the existence of a cyclic *master* or *parent* process, and a set of noncyclic *child* or *slave* processes amongst which work is divided. With this characterization, a very natural correspondence emerges between the processes and the automata which model them: the cyclic *parent* process is very naturally modeled by an ω-automaton, and the *child* processes by a standard finite automaton. Our main contribution of this paper, then, is twofold: first, a formal method of modeling the respective processes in this manner, combining these into a single hybrid system of parallel automata, and secondly, a simple case study demonstrating a practical application of this system. We should note that the notion of parallel finite automata is not a new one; variants have been studied before (e.g. [6, 14]). We take the novel approach of combining *timed* variants ([1, 5]) of finite automata into a single hybrid model which captures the timing properties of the various component processes of a parallel system.

The rest of the paper proceeds as follows: Section 2 defines the automaton models used by our system: Timed Finite Automata in Section 2.1, Timed Büchi Automata in Section 2.2, and a hybrid system combining these two models in Section 2.3. Section 3 gives a case study in the form of an example real-time matrix multiplication kernel, running on a small, four-node real-time parallel cluster. Section 4 concludes.

2 Formalisms

In this section, we give formal definitions for the machinery used in our hybrid system of automata. The definitions given in Sections 2.1 and 2.2 are not new [1]. However, it is still important that we state their definitions here, as they are used later on, in Section 2.3.

2.1 Timed Finite Automata

In this section, we define a simple timed extension of traditional finite state automata and the words they accept. We will use these in later sections to model the timing properties of child processes in a real-time cluster system.

Timed strings take the form $(\bar{\sigma}, \bar{\tau})$, where $\bar{\sigma}$ is a string of symbols, and $\bar{\tau}$ is a monotonically increasing sequence of reals (timestamps). τ_x denotes the timestamp at which symbol σ_x occurs. We also use the notation (σ_x, τ_x) to denote a particular symbol/timestamp pair. For instance, the timed string $((abc), (1, 10, 11))$ is equivalent to the sequence $(a, 1)(b, 10)(c, 11)$, and both represent the case where ‘a’ occurs at time 1, ‘b’ at time 10, and ‘c’ at time 11.

Correspondingly, we extend traditional finite automata to include a set of *timers*, which impose temporal restrictions along state transitions. A timer can be *initialized* along a transition, setting its value to 0 when the transition is taken, and it can be *used* along a transition, indicating that the transition can only be taken if the value of the timer satisfies the specified constraint. Formally, we associate with each automaton a set of timer variables $\overline{T}$, and following the nomenclature of [1], an *interpretation* ν for this set of timers is an assignment of a real value to each of the timers in $\overline{T}$. We write $\nu[T \mapsto 0]$ to denote the interpretation ν with the value of timer T reset to 0. Clock constraints consist of conjunctions of upper bounds:

Definition 1. *For a set $\overline{T}$ of clock variables, the set $X(\overline{T})$ of clock constraints χ is defined inductively as*

$$\chi := (T < c)? \mid \chi_1 \wedge \chi_2$$

where T is a clock in $\overline{T}$ and c is a constant in $\mathbb{R}^+$.

While this definition may seem overly restrictive compared to some other treatments (e.g. [1]), we believe it to be acceptable in this early work for a couple of reasons. First, while simple, this sole syntactic form remains expressive enough to capture an interesting, non-trivial set of use cases (e.g. Section 3). Secondly, the timing analysis in subsequent sections of the paper becomes rather complex, even when timers are limited to this single form. Restricting the syntax in this manner simplifies this analysis to a more manageable level. We leave more complex formulations and the corresponding analysis for future work.

Definition 2 (Timed Finite Automaton (TFA))**.** *A Timed Finite Automaton (TFA) is a tuple*

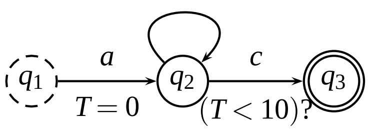
$$\langle \Sigma, Q, s, q_f, \overline{T}, \delta, \gamma, \eta \rangle$$

, where

- *Σ is a finite alphabet,*
- *Q is a finite set of states,*
- *$s \in Q$ is the start state,*
- *$q_f \in Q$ is the accepting state,*
- *$\overline{T}$ is a set of clocks,*
- *$\delta \subseteq Q \times Q \times \Sigma$ is the state transition relation,*
- *$\gamma \subseteq \delta \times 2^{\overline{T}}$ is the clock initialization relation, and*
- *$\eta \subseteq \delta \times X(\overline{T})$ is the constraint relation.*

A tuple $\langle q_i, q_j, \sigma \rangle \in \delta$ indicates that a symbol σ yields a transition from state q_i to state q_j, subject to the restrictions specified by the timer constraints in η. A tuple $\langle q_i, q_j, \sigma, \{T_1, ..., T_n\} \rangle \in \gamma$ indicates that on the transition on symbol σ from q_i to q_j, all of the specified timers are to be initialized to 0. Finally, a tuple $\langle q_i, q_j, \sigma, X(\overline{T}) \rangle \in \eta$ indicates that the transition on σ from q_i to q_j can only be taken if the constraint $X(\overline{T})$ evaluates to true under the current timer interpretation.

Example 3. *The following TFA accepts the timed language $\{(ab^*c, \tau_1 ... \tau_n) \mid \tau_n - \tau_1 < 10\}$ (i.e., the set of all strings consisting of an 'a', followed by an arbitrary number of 'b's, followed by a 'c', such that the elapsed time between the first and last symbols is no greater than 10 time units). The start state is denoted with a dashed circle, and the accepting state with a double line.*



Paths and runs are defined in the standard way:

Definition 4 (Path)**.** *Let* **A** *be a TFA with state set Q and transition relation* δ*. Then* $(q_1,...,q_n)$ *is a* path *over* **A** *if, for all* $1 \leq i < n$*,* $\exists \sigma . \langle q_i, q_{i+1}, \sigma \rangle \in \delta$*.*

Definition 5 (Run)**.** *A run r of a TFA* $\langle \Sigma, Q, q_0, q_f, \overline{T}, \delta, \gamma, \eta \rangle$ *over a timed word* $(\overline{\sigma}, \overline{\tau})$*, is a sequence of the form*

$$r : (q_0, v_0) \xrightarrow[\tau_1]{\sigma_1} (q_1, v_1) \xrightarrow[\tau_2]{\sigma_2} (q_2, v_2) \xrightarrow[\tau_3]{\sigma_3} ... \xrightarrow[\tau_n]{\sigma_n} (q_n, v_n)$$

satisfying the following requirements:

- *Initialization:* $v_0(k) = 0, \forall k \in \overline{T}$
- *Consecution: For all* $i \geq 0$*:*
  - $\delta \ni \langle q_i, q_{i+1}, \sigma_i \rangle$,
  - $(v_{i-1} + \tau_i - \tau_{i-1})$ *satisfies* χ_i*, where* $\eta \ni \langle q_{i-1}, q_i, \sigma_i, \chi_i \rangle$*, and*
  - $v_i = (v_{i-1} + \tau_i - \tau_{i-1})[T \mapsto 0], \forall T \in \overline{T}$*, where* $\gamma \ni \langle q_i, q_j, \sigma_i, \overline{T} \rangle$

r is an accepting *run if* $q_n = q_f$*.*

A TFA **A** *accepts* a timed string $s = (\overline{\sigma}, \tau_1 ... \tau_n)$ if there is an accepting run of s over **A**, and $\tau_n - \tau_1$ is called the *duration* of the string.

Note (Well-Formedness)**.** *We introduce a restriction on how timers can be used in a TFA, thus defining what it means for a TFA to be* well-formed*. Namely, we restrict timers to be used only once along a path; this is to simplify somewhat the timing analysis in subsequent sections. In particular, we say that a TFA* **A** *is well-formed if, for all pairs of states* (q_x, q_y)*, all timers T, and all paths from* q_x *to* q_y*, T is used no more than once. For example, the TFAs shown in Figure 1 are not well-formed, since in both cases, timers can potentially be used more than once— in the first case (*$\mathbf{A}_1$*), along the self-loop on* q_2*, and in the second case (*$\mathbf{A}_2$*), along two separate transitions along the path. At first, this may appear to be overly restrictive, but as it turns out, many of these cases can easily be rewritten equivalently to conform to the single-use restriction, as shown in Figure 2.*

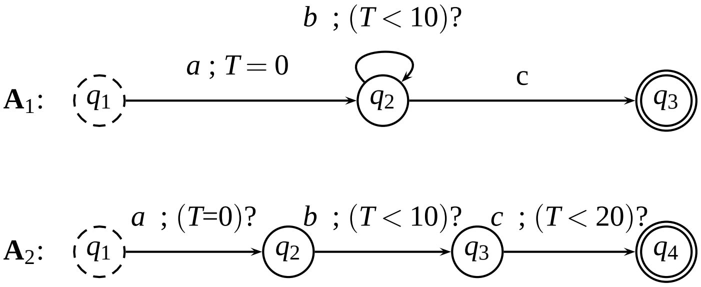


Figure 1: Malformed TFAs. Start states are denoted with a dashed circle, and accepting states with a double line. The intent of $\mathbf{A}_1$ is to allow strings of the form a, followed by arbitrarily many bs, as long as they all occur less than 10 units after the a, followed by a c. The intent of $\mathbf{A}_2$ is to allows strings of the form abc, where the elapsed time between the a and b is less than 10, and that between the a and c is less than 20. Both of these can be rewritten using conforming automata, as shown in Figure 2.

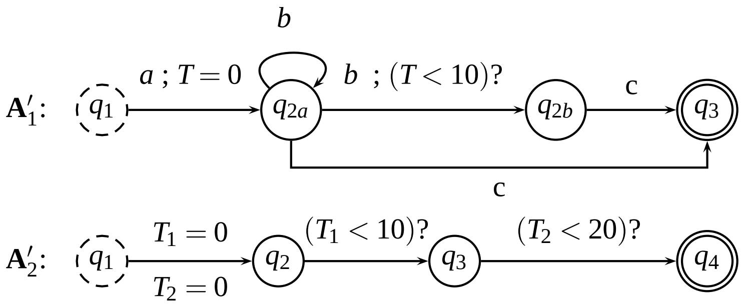


Figure 2: Equivalent, well-formed versions of automata from Figure 1.

2.1.1 Bounding Maximum Delay

An important notion throughout the remainder of the paper is that of computing bounds on the allowable delays along all possible paths through a TFA. Specifically, we are interested in doing so to be able to reason formally about the maximum execution time for a child process, with the end goal of being able to bound the execution time of the system— parent and all child processes— as a whole.

The idea is that we will ultimately use TFAs to represent the timing properties of a child process. Paths through the automaton from its start state to an accepting state correspond to possible execution paths of the child process' code. Certainly, proving a tight upper bound on the delay between two arbitrary points along an execution path remains a very difficult problem, but to be clear, this is not our goal. Rather, our approach involves modeling an execution path through a child process (and, by extension, its corresponding timed automaton) using an *event-based* model, in which selected system events are modeled by transitions in the automaton, and we rely on timing properties of the process to be guaranteed by the underlying RTOS process scheduler. The problem of computing the worst-case delay through the automaton equates to that of computing the maximum delay over all possible paths through the automaton from its start state to its accepting state:

$$\Delta_{\mathbf{A}} = \max_{p \in \mathbf{paths}(\mathbf{A})} \Delta(p)$$

where

- $\mathbf{A} = \langle \Sigma, Q, q_0, q_f, \overline{T}, \delta, \gamma, \eta \rangle$ is the TFA
- $\mathbf{paths}(\mathbf{A})$ denotes the set of all paths in $\mathbf{A}$ from its start state q_0 to accepting state q_f, and
- $\Delta(p)$, for path $p = (q_0, ..., q_f)$, denotes the maximum delay from q_0 to q_f. That is, the maximum duration of any timed string $(\overline{\sigma}, \overline{\tau})$ such that $(q_0 ... q_f, \overline{v})$ is a run of the string over $\mathbf{A}$ (for some $\overline{v}$).

This problem can thus be formulated in the following manner: given a timed finite automaton **A** and an integer n, is there a timed word of duration $d \geq n$ that is accepted by **A**? While simple cases, such as those presented in this paper, can be computed by observation and enumeration, the complexity of the general problem remains an open question, although we highly suspect it to be intractable— Courcoubetis and Yannakakis give exponential-time algorithms for this and related problems, and have shown a strictly more difficult variant of the problem to be **PSPACE**-complete [4]. Furthermore, expanding the timer constraint syntax to a more expressive variant (c.f. [1]) can only complicate matters in terms of complexity. We must be cautious, then, to ensure that we do not impose an inordinately large number of timers on a child process.

2.2 Timed Büchi Automata

Whereas we model the timing properties of the child processes of a cluster system using the timed finite automata of the previous section, we model these properties of the parent using a timed variant of ω-automata, specifically Timed Büchi Automata. We assume a basic familiarity with these; due to space constraints, we give only brief overview here. To review briefly, ω-automata, like standard finite automata, also consist of a finite number of states, but instead operate over words of infinite length. Classes of ω-automata are distinguished by their acceptance criteria. Büchi automata, which we consider in this paper, are defined to accept their input if and only if a run over the input string visits an accepting state infinitely often. Other classes of ω-automata exist as well. For example, Muller automata are more stringent, specifying their acceptance criteria as a *set* of acceptance sets; a Muller automaton accepts its input if and only if the set of states visited infinitely often is specified as an acceptance set. More detailed specifics can be found elsewhere— for example, [1].

A *Timed Büchi Automaton* (TBA) is a tuple $\langle \Sigma, Q, q_0, q_f, \overline{T}, \delta, \gamma, \eta \rangle$, where

- Σ is a finite alphabet,
- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is a set of accepting states,
- $\overline{T}$ is a set of clocks,
- $\delta \subseteq Q \times Q \times \Sigma$ is the state transition relation,
- $\gamma \subseteq \delta \times 2^{\overline{T}}$ is the clock initialization relation, and
- $\eta \subseteq \delta \times X(\overline{T})$ is the constraint relation.

A tuple $\langle q_i, q_j, \sigma \rangle \in \delta$ indicates that a symbol σ yields a transition from state q_i to state q_j, subject to the restrictions specified by the clock constraints in η. A tuple $\langle q_i, q_j, \sigma, \overline{T} \rangle \in \gamma$ indicates that on the transition on symbol σ from q_i to q_j, all clocks in $\overline{T}$ are to be initialized to 0. Finally, a tuple $\langle q_i, q_j, \sigma, X(\overline{T}) \rangle \in \eta$ indicates that the transition on σ from q_i to q_j can only be taken if the constraint $X(\overline{T})$ evaluates to true under the values of the current timer interpretation.

We define *paths*, *runs*, and *subruns* over a TBA analagously to those over a TFA:

Definition 6 (Path (TBA)). *Let $\mathscr{A}$ be a TBA with state set Q and transition relation δ. $(q_1, ..., q_n)$ is a* path *over* **A** *if, for all* $1 \leq i < n$, $\exists \sigma . \langle q_i, q_{i+1}, \sigma \rangle \in \delta$.

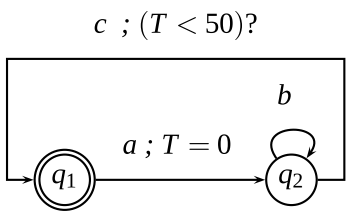
Definition 7 (Run, Subrun (TBA)). *A run (subrun) r, denoted by $(\overline{q}, \overline{v})$, of a Timed Büchi Automaton $\langle \Sigma, Q, q_0, q_f, \overline{T}, \delta, \gamma, \eta \rangle$ over a timed word $(\overline{\sigma}, \overline{\tau})$, is an infinite (finite) sequence of the form*

$$r : (q_0, v_0) \xrightarrow[\tau_1]{\sigma_1} (q_1, v_1) \xrightarrow[\tau_2]{\sigma_2} (q_2, v_2) \xrightarrow[\tau_3]{\sigma_3} \dots$$

satisfying the same requirements as given in Definition 5.

For a run r, the set $inf(r)$ denotes the set of states which are visited infinitely many times. A TBA $\mathscr{A}$ with final states F accepts a timed word $w = (\overline{\sigma}, \overline{\tau})$ if $inf(r) \bigcap F \neq \emptyset$, where r is the run of w on $\mathscr{A}$. That is, a TBA accepts its input if any of the states from F repeat an infinite number of times in r.

Example 8. *Consider the following TBA $\mathscr{A}_1$, with start state q_1 and accept states $F = \{q_1\}$:*



This TBA accepts the ω-language $L_1 = \{((ab^*c)^\omega, \tau) \mid \forall x.\exists i, j.\forall k.\phi\}$ *where* ϕ *is the boolean formula*

$$\tau_i < \tau_k < \tau_j \implies (\sigma_i = a) \wedge (\sigma_k = b) \wedge (\sigma_j = c) \wedge (\tau_j - \tau_i < 50)$$

Lastly, we take the concept of maximum delay, introduced in the previous section with respect to Timed Finite Automata, and extend it to apply to Timed Büchi Automata. Doing so first requires the following definition, which allows us to restrict the timing analysis for TBAs to finite subwords:

Definition 9 (Subword over $\overline{q}$)**.** *Let* $\mathscr{A}$ *be a TBA, and let* $\overline{q} = (q_m...q_n)$ *be a finite path over* $\mathscr{A}$*. A finite timed word* $w = ((\sigma_m...\sigma_n), (\tau_m...\tau_n))$ *is a* subword over $\overline{q}$ *iff* $\exists q_0, ..., q_{m-1}, \sigma_0, ..., \sigma_{m-1}, \tau_0, ..., \tau_{m-1}$ *such that* $(q_0...q_{m-1}q_m...q_n, \overline{v})$ *is a subrun of* $((\sigma_0...\sigma_{m-1}\sigma_m...\sigma_n), (\tau_0...\tau_{m-1}\tau_m...\tau_n))$ *over* $\mathscr{A}$ *for some* $\overline{v}$*.*

Definition 9 is a technicality which is necessary to support the following definition of the maximum delay between states of a TBA:

Definition 10. *Let* $\mathscr{A}$ *be a TBA, and let* $\overline{q}$ *be a finite path over* $\mathscr{A}$*. Then* $\Delta_{\mathscr{A}}(\overline{q})$ *is the maximum duration of any subword over* $\overline{q}$*.*

Example 11. *Consider* $\mathscr{A}_1$ *from Example 8. Then* $\Delta_{\mathscr{A}_1}(q_1q_2q_2q_1) = 50$*.*

Algorithmically computing $\Delta_{\mathscr{A}}(\overline{q})$ for a TBA $\mathscr{A}$ is analogous to the case for TFAs; in small cases (i.e., relatively few timers with small time constraints), the analysis is relatively simple, while we conjecture the problem for more complex cases to be intractable; we leave more detailed analysis for future work.

2.3 Parallel Timing Systems

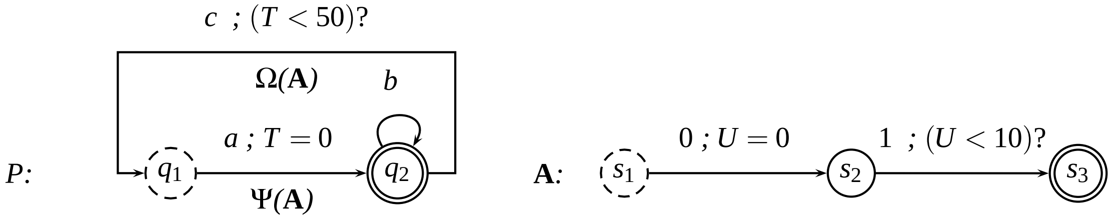
Next, we model the timing properties of a SPMD-type parallel system as a whole by combining the two models of Sections 2.1 and 2.2 into a single *parallel timing system*. A *parallel timing system* (PTS) is a tuple $\langle P, \overline{\mathbf{A}}, \psi, \varphi \rangle$, where

- $P = \langle \Sigma, Q, q_0, q_f, \overline{T}, \delta, \gamma, \eta \rangle$ is a TBA (used to model the timing properties of the parent process)
- $\overline{\mathbf{A}}$ is a set $\{\mathbf{A}_1, ..., \mathbf{A}_n\}$ of TFAs (used to model the timing properties of the child processes)
- $\psi \subseteq \delta \times \overline{\mathbf{A}}$ is a *fork* relation (used to model the spawning of child processes)
- $\varphi \subseteq \delta \times \overline{\mathbf{A}}$ is a *join* relation (used to model barriers (joins))

A tuple $\langle q_i, q_j, \sigma, \mathbf{A} \rangle$ in ψ, with $\mathbf{A} \in \overline{\mathbf{A}}$, indicates that an instance of $\mathbf{A}$ is to be "forked" on the transition from q_i to q_j on symbol σ, and this "fork" is denoted graphically as $q_i \xrightarrow{\Psi(\mathbf{A})} q_j$, modeling the spawning of a child process along the transition. Similarly, a tuple $\langle q_i, q_j, \sigma, \mathbf{A} \rangle$ in φ indicates that a previously forked instance of $\mathbf{A}$ is to be "joined" on the transition from q_i to q_j on symbol σ. This "join" is denoted graphically as $q_i \xrightarrow{\Omega(\mathbf{A})} q_j$, modeling the joining along the transition with a previously spawned child process[1].

[1] Ψ was chosen as the symbol for 'fork', as it graphically resembles a "fork"; Ω was chosen as that for 'join', as it connotes "ending" or "finality".

Example 12. *Consider the following timing system $S_1 = \langle P, \{\mathbf{A}\}, \psi, \varphi\rangle$:*



P is the parent TBA with initial state q_1 and final state set $F = \{q_2\}$. P accepts the ω-language L_1 (see p. 45), and $\mathbf{A}$ is a TFA which accepts the timed language $\{(01, \tau_1\tau_2) \mid \tau_2 - \tau_1 < 10\}$. In addition, the fork *and* join *relations ψ and φ dictate that on the transition from q_1 to q_2, an instance of $\mathbf{A}$ is forked ($\Psi(\mathbf{A})$), and that the transition from q_2 to q_1 can only proceed once that instance of $\mathbf{A}$ has completed ($\Omega(\mathbf{A})$).*

Conceptually, this system models a parent process (P) which exhibits periodic behavior, accepting an infinite number of substrings of the form ab^*c, in which the initial 'a' triggers a child process $\mathbf{A}$ which must be completed prior to the end of the sequence, marked by the following 'c'. In addition, the 'c' must occur no more than 50 time units after the initial 'a'. The child process is modeled by $\mathbf{A}$, which accepts strings of the form 01, *in which the* 1 *must occur no more than ten time units after the initial* 0.

In theory, child processes could spawn children of their own (e.g. recursion). For now, however, we disallow this possibility, as it somewhat complicates the analysis in the following section without adding significantly to the expressive power of the model. The model can be expanded later to allow for arbitrarily nested children of children with the appropriate modifications; specifically, TBAs would need to be extended to include their own ψ and φ relations, as would the definition of Δ for TBAs.

Before proceeding, it is important to note that a PTS $S = \langle P, \overline{\mathbf{A}}, \psi, \varphi\rangle$ is not itself interpreted as an automaton. In particular, we do not ever define a language accepted by S. Indeed, it is not entirely clear what such a language would be, as we never specify the input to any of the children in $\mathbf{A}$. Rather, the sole intent in specifying such a system S is to specify the *timing behavior* of the overall system, rather than any particular language that would be accepted by it.

2.3.1 Consistency

With this said, we note that in Example 12, $\mathbf{A}$ is in some sense "consistent" with its usage in P. Specifically, since the maximum duration of any string accepted by $\mathbf{A}$ is 10, we are guaranteed that any instance of $\mathbf{A}$ forked on the $q_1 \xrightarrow{a} q_2$ transition will have completed in time for the 'join' along the $q_2 \xrightarrow{c} q_1$ transition and hence, the timer $(T < 50)$? on this transition would be respected in all cases. In this sense, all $(\Psi(\mathbf{A}), \Omega(\mathbf{A}))$ pairs are consistent with timer T. However, such consistency is not always the case. Consider, for instance, the parallel timing system S_2 shown in Figure 3. In this case, there are two child

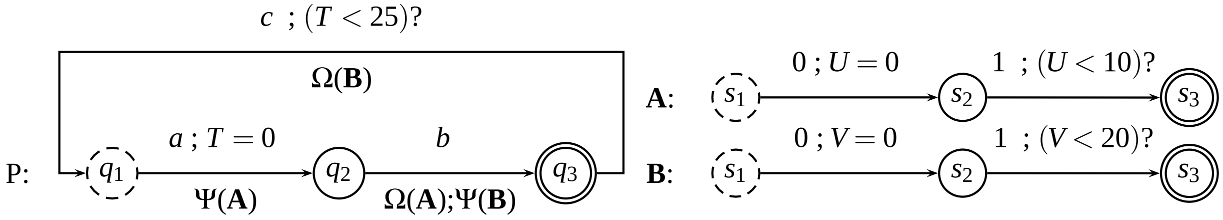


Figure 3: An inconsistent parallel timing system S_2.

processes: **A** and **B**. The maximum duration of a timed word accepted by **A** is 10, and that of **B** is 20. Supposing that an ‘a’ occurs (and **A** forked) at time 0, it is thus possible that the **A** will not complete until time $10 - \varepsilon_1$, at which time the ‘b’ and fork of **B** can proceed. It is therefore possible that **B** will not complete until time $30 - \varepsilon_1 - \varepsilon_2$ (for small $\varepsilon_1, \varepsilon_2$). This would then violate the $(T < 25)$? constraint, corresponding to a case in which a child process could take longer to complete than is allowable, given the timing constraints of the parent process. It is precisely this type of interference which we must disallow in order for a timing system to be considered consistent with itself.

To this end, we propose a method of defining *consistency* within a timing system. Informally, we take the approach of deriving a new set of conditions from the timing constraints of the child processes, so that checking *consistency* reduces to the process of verifying that these conditions respect the timing constraints of the master process.

First, we replace $\bar{\mathbf{A}}$, ψ, and φ from the parallel timing system with a new set of *derived* timers, one for each $\mathbf{A} \in \bar{\mathbf{A}}$, defining the possible “worst case” behavior of the child processes. Each such timer $T_\mathbf{A}$ is initialized on the transition along which the corresponding **A** is forked, and is used along (constrains) any transitions along which **A** is joined. Each such use ensures that the timer is less than $\Delta_\mathbf{A}$, representing the fact that the elapsed time between the forking and joining of a child process is bounded in the worst case by $\Delta_\mathbf{A}$— the longest possible duration for the child process. As an example, “flattening” the timing system S_1 of Example 12 results in a single new timer $T_\mathbf{A}$, initialized along the $q_1 \xrightarrow{a} q_2$ transition, and used along the $q_2 \xrightarrow{c} q_1$ transition with the constraint $(T_\mathbf{A} < 10)$?. We then check that none of these new derived timers invalidate the timing constraints of the parent process.

Formally, we define two relations. The first of these is *flattening*, which takes a parallel timing system $\langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ and yields a new pair of relations (γ, η). Intuitively, γ defines the edges along which each of the derived timers are initialized, and η defines the edges along which each of the derived timers are used:

Definition 13. *Let* $S = \langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ *be a parallel timing system. Then* $\mathbf{flatten}(S) = (\gamma, \eta)$*, where*

$$\gamma = \{\langle q_i, q_j, \sigma, \{T_\mathbf{A}\}\rangle \mid \langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \psi\}$$
$$\eta = \{\langle q_i, q_j, \sigma, X\rangle \mid \langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \varphi\}$$

and

$$X = \bigwedge_{\langle q_i, q_j, \sigma, \mathbf{A}\rangle \in \varphi} (T_\mathbf{A} < \Delta_\mathbf{A})$$

The second relation takes ψ and φ as inputs and extracts a set of edge pairs, defined such that each such pair (e_1, e_2) specifies when a derived timer is initialized (e_1) and used (e_2).

Definition 14. *Let* $S = \langle P, \bar{\mathbf{A}}, \psi, \varphi \rangle$ *be a parallel timing system, with* $\mathbf{A} \in \bar{\mathbf{A}}$*. Then the set of all use pairs of* **A** *in* S *is defined as* $\mathbf{pairs}(\mathbf{A}, S) = \{((q_x, q_y), (q_m, q_n)) \mid (\langle q_x, q_y, \sigma_1, \mathbf{A}\rangle \in \psi) \wedge (\langle q_m, q_n, \sigma_2, \mathbf{A}\rangle \in \varphi)\}$ *for some* σ_1, σ_2*. Furthermore,*

$$\mathbf{pairs}(S) = \bigcup_{\mathbf{A} \in \bar{\mathbf{A}}} \mathbf{pairs}(\mathbf{A}, S)$$

Example 15. *Consider parallel timing system* S_3 *shown in Figure 4. Observe that* $\Delta_\mathbf{A} = 25$ *and* $\Delta_\mathbf{B} = 11$*. Then:* $\mathbf{flatten}(S_3) = (\gamma, \eta)$*, where*

$$\gamma = \{\langle q_1, q_2, a, \{T_\mathbf{A}\}\rangle, \langle q_2, q_3, b, \{T_\mathbf{B}\}\rangle\}$$
$$\eta = \{\langle q_3, q_1, c, X\rangle\} \text{ , where } X = (T_\mathbf{A} < 25) \wedge (T_\mathbf{B} < 11)$$

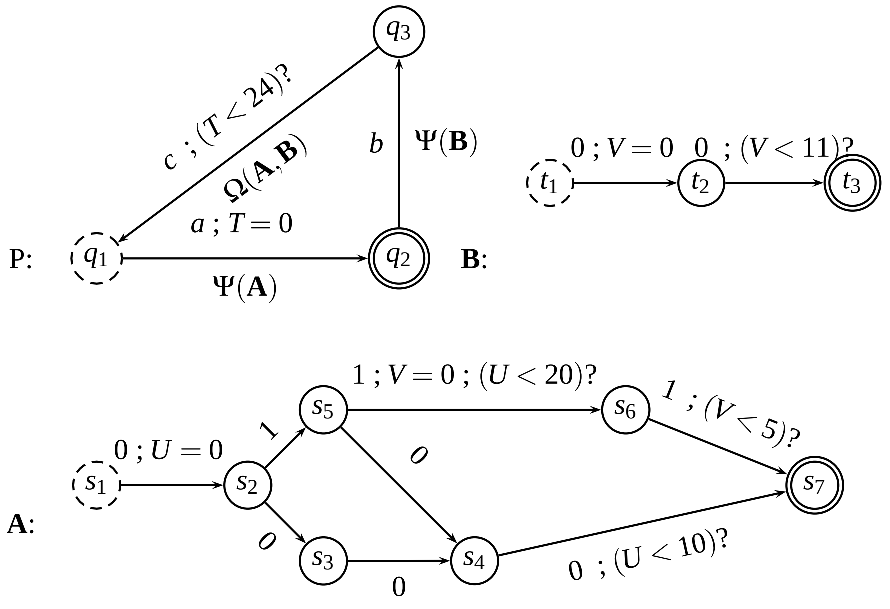


Figure 4: Parallel timing system S_3. $\Delta_{\mathbf{A}} = 25, \Delta_{\mathbf{B}} = 11$.

shown graphically in Figure 5, and

$$\begin{aligned}\mathbf{pairs}(S) &= \mathbf{pairs}(\mathbf{A},S) \cup \mathbf{pairs}(\mathbf{B},S)\\ &= \{((q_1,q_2),(q_3,q_1))\} \cup \{((q_2,q_3),(q_3,q_1))\}\\ &= \{((q_1,q_2),(q_3,q_1)),((q_2,q_3),(q_3,q_1))\}\end{aligned}$$

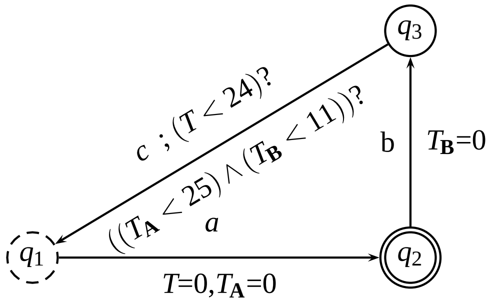


Figure 5: The result of flattening S_3: forks and joins of **A** and **B** are shown along with derived timers $T_{\mathbf{A}}$ and $T_{\mathbf{B}}$. Compare with Figure 4.

We can now proceed with a formal definition of consistency for a parallel timing system. Recall that intuitively, such a system is consistent if the worst case timing scenarios over all child processes will not invalidate the timing constraints of the parent process— in other words, if the maximum delay between two states allowed by the child processes never exceeds the corresponding maximum delay allowed by the timers in the parent process.

Definition 16 (Consistency)**.** *Let* $S = \langle \mathscr{A}, \psi, \varphi, \overline{\mathbf{A}} \rangle$ *be a PTS, where*

- $\mathscr{A} = \langle \Sigma, Q, q_0, F, \overline{T}, \delta, \gamma, \eta \rangle$ *is a TBA*
- $\mathbf{flatten}(S) = (\gamma', \eta')$

- $\mathscr{A}' = \langle \Sigma, Q, q_0, F, \overline{T}, \delta, \gamma', \eta' \rangle$

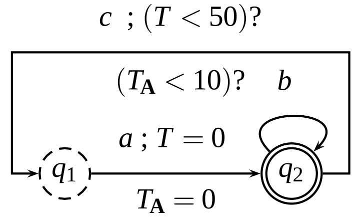
Then S is consistent *if for all edge pairs $((q_x, q_y), (q_m, q_n)) \in$ **pairs**(S) and all paths $p = q_x q_y ... q_m q_n$ through $\mathscr{A}$,*

$$\Delta_{\mathscr{A}'}(p) \leq \Delta_{\mathscr{A}}(p) \tag{16.1}$$

We conclude this section with a few simple examples, which should help to clarify Definition 16; the following section gives a more realistic example.

Example 17. *S_1 is consistent.*

Proof. P', the result of flattening S_1, is shown below, with $T_{\mathbf{A}}$ being the derived timer corresponding to **A**:



Furthermore, **pairs**$(P') = \{((q_1, q_2), (q_2, q_1))\}$, and by observation, all paths through P' beginning with the edge (q_1, q_2) and ending with the edge (q_1, q_2) take the form $q_1(q_2)^*q_1$. All such paths p satisfy inequality 16.1, and thus by definition, S_1 is consistent. □

Example 18. *S_3 is not consistent.*

Proof. P', the result of flattening S_3, is shown in Figure 5. Furthermore,

$$\mathbf{pairs}(P') = \{((q_1, q_2), (q_3, q_1)), ((q_2, q_3), (q_3, q_1))\}$$

There are thus two paths against which we need to test inequality 16.1: $(q_1 q_2 q_3 q_1)$, and $(q_2 q_3 q_1)$; the first of these fails the test: $\Delta_{P'}(q_1 q_2 q_3 q_1) = 25$,and $\Delta_P(q_1 q_2 q_3 q_1) = 24$.

□

3 Case Study: Matrix Multiplication

We now turn our attention to a practical application of the concepts discussed so far. Namely, we demonstrate the use of the formal validation concepts on a simple parallel, MPI-style [13, 15] matrix multiplication kernel, extracted from the larger power-grid analysis application described in [11, 12]. Our kernel implements a variant of Fox's algorithm for matrix multiplication [7]. For simplicity, we assume square matrices, and that the number of columns, rows, and processors are all perfect squares. The algorithm distributes the task of multiplying two matrices amongst all processors in the system.

We give a simple distributed algorithm for matrix multiplication, and a consistent parallel timing system for that algorithm. We conclude the section with empirical results— timing measurements taken on a small, four-node real-time cluster, each node consisting of dual quad-core 2.66Ghz Xeon X5660 processors running the Xenomai RTOS with 48GB RAM. The timing measurements of the PTS, along with the usual restrictions associated with real-time computation (e.g. no virtual memory or paging, process scheduling, ensuring minimal variance in execution timings, etc.), are bounded by virtue of

Xenomai's real-time process scheduler. The result is a matrix multiplication kernel which provably runs in under 9 ms per cycle for 128×128 double-precision matrices. We emphasize that we are not claiming the speed of the operation to be a groundbreaking result— obviously, this is a relatively small matrix size, but was so chosen as this is the order of the size required by our targeted application kernel. Rather, we give these numbers, as well as the PTS, to illustrate the *process* by which we analyze the temporal interactions between processes, thus showing this delay to be a provable upper bound.

3.1 Algorithm

Algorithm 1 MatrixMultiply: Compute $\mathbf{C} = \mathbf{A} \times \mathbf{B}$

```
$p$ : Number of processors
$N$ : Rank of matrices
$q \leftarrow \sqrt{p}$
while true do
  $dest \leftarrow 1$
  if self $==$ 0 then {Master process}
    for $i = 0$ to $q - 1$ do
      for $j = 0$ to $q - 1$ do
        $w \leftarrow i\frac{N}{q}, x \leftarrow (i+1)\frac{N}{q}$
        $y \leftarrow j\frac{N}{q}, z \leftarrow (j+1)\frac{N}{q}$
        $\overline{X} \leftarrow \mathbf{A}[w:x][0:N]$
        $\overline{Y} \leftarrow \mathbf{B}[0:N][y:z]$
        if $i \neq 0$ and $j \neq 0$ then {Master already has these chunks}
          send$(\overline{X}, dest)$
          send$(\overline{Y}, dest)$
          $dest \leftarrow dest + 1$
        end if
      end for
    end for
  else {Child processes}
    $\overline{X} \leftarrow$ recv$(0)$
    $\overline{Y} \leftarrow$ recv$(0)$
  end if
  $\overline{Z} \leftarrow$ locMM$(\overline{X}, \overline{Y})$
  reduce$(\overline{Z}, \mathbf{C})$
end while
```

The pseudocode for the algorithm is given in Algorithm 1. Conceptually, to multiply two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ using a p processor cluster, each matrix is divided into segments, which are then distributed in round-robin fashion amongst the processors of the cluster. Each processor then performs a local matrix multiplication on its own local submatrices, and the results of these local operations are aggregated (reduced) to form the matrix product $\mathbf{A} \times \mathbf{B}$.

Due to space constraints, we will not describe the partitioning in detail; Figure 6 shows the partitioning and distribution of work by Algorithm 1 for a four-processor cluster. In this figure, $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are all $N \times N$ matrices. $\mathbf{A}$ is partitioned into 2 sets of $\frac{N}{2}$ rows each, and $\mathbf{B}$ is partitioned into 2 sets of $\frac{N}{2}$

columns each. The master process, p_0, computes the local product $\mathbf{A}_1 \times \mathbf{B}_1$, and writes the result to $\mathbf{C}_1$. p_0 then sends submatrices $\mathbf{A}_1$ and $\mathbf{B}_2$ to p_1, who then computes their product, writing the result to $\mathbf{C}_2$. Similarly, p_2 receives and computes $\mathbf{C}_3 = \mathbf{A}_2 \times \mathbf{B}_1$, and p_3 receives and computes $\mathbf{C}_4 = \mathbf{A}_2 \times \mathbf{B}_2$.

The algorithm proceeds as follows: the master process executes lines 5 through 17, which partition **A** and **B** into submatrices (lines 7–10), and send these parts out to the respective child processes (lines 12–13). Conversely, the child processes execute lines 19–20, which receive the submatrices assigned by the master process. Lines 22–23 are run by all processes, including the master process (which, in this case, participates in the task of matrix multiplication as well). Line 22 performs the local operation, line 23 writes the local result to the appropriate location in **C**.The entire process then repeats indefinitely, as given by the **while** loop (lines 2 and 24).

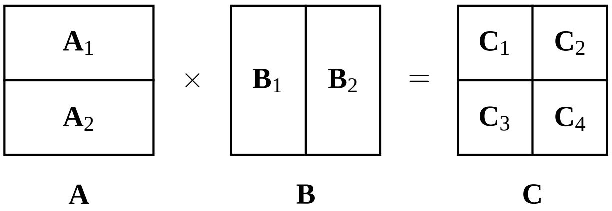


Figure 6: Partitioning and distribution of matrix multiplication by Algorithm 1 across a four processor cluster.

3.2 Parallel Timing System

Figure 7 shows a parallel timing system for Algorithm 1 across a four processor cluster, consisting of the TBA P_{MM}, which models the master process, and a child TFA $\mathbf{A}_{MM}$, modeling instances of the child processes. Specific events have been elided from the diagram in this case, since events in this case always represent transitions between statements.

3.2.1 Parent

States in the parent automaton P are prefixed with a 'P', followed by the line number as given in Algorithm 1. For example, $P3$ corresponds to the state of the parent process as it is executing line 3.

Additionally, lines 12 and 13 each beget three separate states— parameterized on the values of the loop induction variables i and j— and are labeled accordingly. As is commonly the case in WCET analysis, unrolling the loop nest in this fashion is necessary in order to obtain a strict upper bound on the number of iterations and, consequently, the total execution time, of the loop nest.

P forms, in this case, a simple cycle. The cycle starts at state $P3$, and steps sequentially through the steps (states) of the algorithm. Namely, the parent process starts at line 3 (i.e., state $P3$), and proceeds sequentially through lines 4 (state $P4$), and eventually to line 12 ($P12^{i=0}_{j=1}$). The delay between the initialization (state $P3$) and the first send ($P12^{i=0}_{j=1}$) is bounded by a timer, T_{setup1} (the idea being that this is the delay incurred by the time to "set up" the first send). Execution then proceeds to line 13 ($P13^{i=0}_{j=1}$); the delay along this transition represents the time to send the first chunk to the respective child process, and is bounded by timer T_{send1}. At this point, execution proceeds to line 14 ($P14^{i=0}_{j=1}$). Along this transition, there are two items to note: first, the time to process the second send is bounded by the timer T_{send2}, and second, the child process has now been sent the data it needs, and consequently, $\mathbf{A}_1$ is forked. Execution proceeds similarly through the next six states, representing the unwound iterations of the loop nest. Child process $\mathbf{A}_2$ is similarly forked on the transition from $P13^{i=1}_{j=0}$ to $P14^{i=1}_{j=0}$, and $\mathbf{A}_3$ on the transition from $P13^{i=1}_{j=1}$ to $P14^{i=1}_{j=1}$. Execution then proceeds through lines 22 (state $P22$) and 23 ($P23$). The duration of the local matrix multiplication operation (line 22) is bounded by the timer T_{MM}, and that through the reduce operation (line 23) by the timer T_{reduce}. Additionally, the transition from $P23$ back to $P3$ waits for (joins with) all child processes to complete before proceeding.

3.2.2 Child

In this case, the child processes are modeled by the TFA **A**. Nomenclature is analogous to that of P: states in **A** are prefixed with an **A**, followed by the corresponding line number from Algorithm 1.

The child process starts at line 18 (state **A**18). The process then proceeds to receive the first block of data (line 19, state **A**19). The time to process the receive is bounded by timer T_{recv1}. Execution proceeds to receive the second block of data (line 20, state **A**20). The time to process this second receive is bounded by T_{recv2}. Execution proceeds next to the local matrix multiplication (line 22, state **A**22); the time spent on this operation is bounded by timer T_{locMM}. Finally, execution proceeds to the data writeback (line 23, state **A**23); the time spent on this operation is bounded by timer T_{reduce}.

Theorem 19. *S_{MM} is consistent.*

Proof. Let $\mathbf{flatten}(S_{MM}) = (\gamma', \eta')$, with $P'_{MM} = \langle \Sigma, Q, q_0, F, \overline{T}, \delta, \gamma', \eta' \rangle$. By definition, $\mathbf{pairs}(S_{MM}) = \{(e_1, e_4), (e_2, e_4), (e_3, e_4)\}$, where

$$e_1 = (P13^{i=0}_{j=1}, P14^{i=0}_{j=1}) \qquad e_2 = (P13^{i=1}_{j=0}, P14^{i=1}_{j=0})$$
$$e_3 = (P13^{i=1}_{j=1}, P14^{i=1}_{j=1}) \qquad e_4 = (P23, P3)$$

By observation, there are three paths which we must consider:

$$p_1 = (P13^{i=0}_{j=1} P14^{i=0}_{j=1} ... P23P3)$$
$$p_2 = (P13^{i=1}_{j=0} P14^{i=1}_{j=0} ... P23P3)$$
$$p_3 = (P13^{i=1}_{j=1} P14^{i=1}_{j=1} ... P23P3)$$

The rest of the proof follows by enumeration:

$$(\Delta_{P'_{MM}}(p_1) = 6.1) < (\Delta_{P_{MM}}(p_1) = 8.1)$$
$$(\Delta_{P'_{MM}}(p_2) = 6.1) < (\Delta_{P_{MM}}(p_2) = 7.3)$$
$$(\Delta_{P'_{MM}}(p_3) = 6.1) < (\Delta_{P_{MM}}(p_3) = 6.5)$$

□

Finally, we note that the worst case delay along one iteration of the algorithm is 8.9 ms. This follows from the observation that the parent automaton P takes the form of a simple cycle with no unbound segments (i.e., subpaths which are not constrained by any timer). Specifically, P consists of consecutive pairs of segments, each constrained by pairs of timers. Consequently, we can derive an upper bound for a single iteration of the algorithm by summing the bounds of all of the timers, yielding the specified upper bound. Combined with Theorem 19, which ensures that the timing of the child processes does not invalidate this bound, we are left with a cyclic, parallel, time-bounded matrix multiplication kernel.

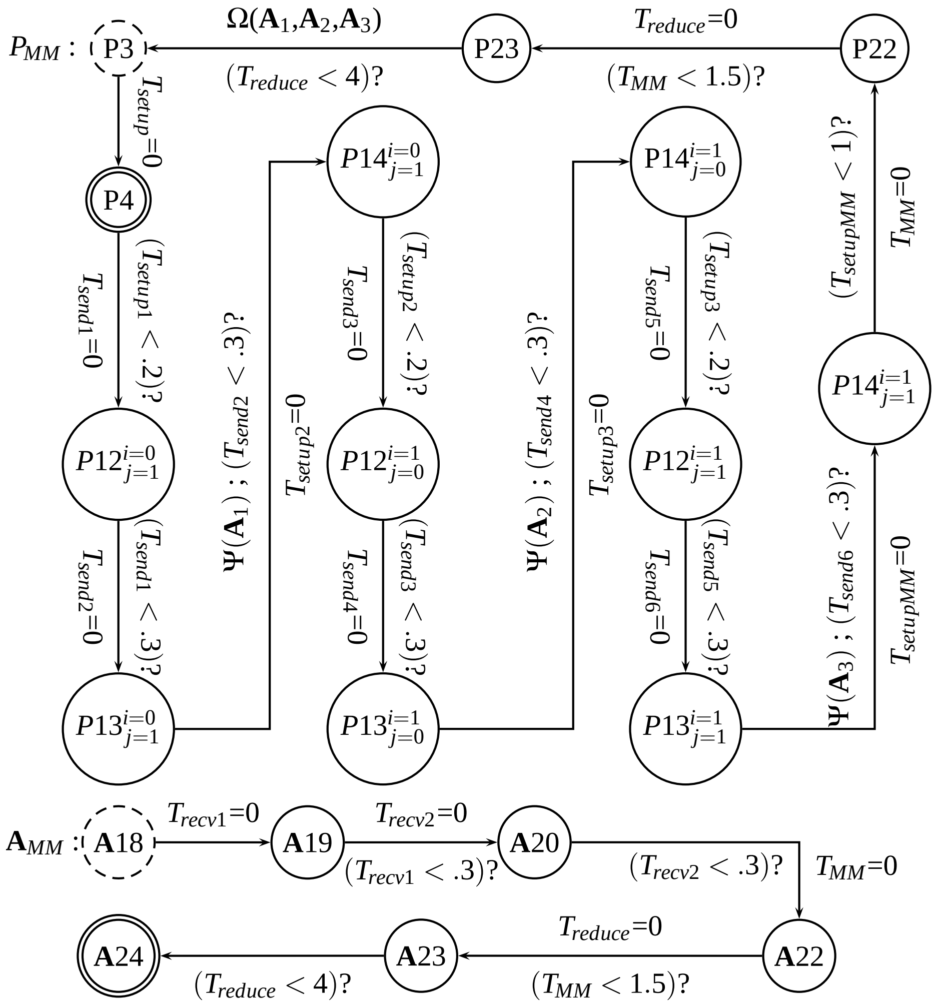


Figure 7: Parallel timing system S_{MM} for Algorithm 1 across a four processor cluster. Events have been elided for the sake of clarity. Upper bounds on timer constraints correspond to delay measurements taken over our implementation; times are given in milliseconds. Minimal variance from these bounds is ensured to the extent provided by the underlying RTOS.

4 Concluding Remarks

We conclude with a few closing remarks. We have presented a formal system for modeling the temporal properties of a restricted class of real-time parallel systems, with a simple example of an application kernel. As is usually the case with real-time systems, loops need to be unrolled, bounding the number of iterations, in order to obtain an upper bound on the total execution time of the loop. Algorithm 1 (intentionally) distills to a relatively simple PTS, due to the basic structure of the control flow graph of both the parent and child processes; more complex examples are of obvious interest for future work. Similarly, the model in Figure 7 in our case was derived manually— in this case, a relatively simple task. More complex examples can certainly prove to be more of a challenge, and automated tools for this task are desirable. One possible approach for such automation would be compiler-driven, whereby users could specify to the compiler (via `#pragmas`, for instance), events of interest, and the compiler could proceed to output the appropriate annotated control flow graph.

We assume timing behavior is consistent across all child processes, although if there were to be significant variance across child processes (e.g. heterogeneous or NUMA architectures) we could account for such behavior using different child TFAs.

Additionally, we have laid out several interesting open questions which arise out of the analysis of our relatively straightforward formulation: what is the complexity of computing the worst case delay along a single path of a TFA (TBA), and through a TFA (TBA) in general? Up to this point, we have only considered conjunctions of maximum constraints; how does this change in the presence of a more generalized constraint syntax (c.f. [1])?

We have largely been working with the SPMD execution model paradigmatic of many MPI-type programs. It would be interesting to investigate temporal models for other parallel models (e.g. OpenMP) as well. Lastly, our application kernel distills to a relatively simple set of automata. More complex examples are certainly of interest, and are on the horizon for future work.

References

[1] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theor. Comput. Sci.* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] Fang Chen, Xueshan Han, Zhiyuan Pan & Li Han (2008): *State Estimation Model and Algorithm Including PMU*. In: *Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008. Third International Conference on*, pp. 1097 –1102, doi:10.1109/DRPT.2008.4523571.

[3] Yousu Chen, Zhenyu Huang & D. Chavarria-Miranda (2010): *Performance evaluation of counter-based dynamic load balancing schemes for massive contingency analysis with different computing environments*. In: *Power and Energy Society General Meeting, 2010 IEEE*, pp. 1 –6, doi:10.1109/PES.2010.5589536.

[4] Costas Courcoubetis & Mihalis Yannakakis (1992): *Minimum and maximum delay problems in real-time systems*. *Form. Methods Syst. Des.* 1(4), pp. 385–415, doi:10.1007/BF00709157.

[5] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman & Andrzej Wasowski (2010): *Timed I/O automata: a complete specification theory for real-time systems*. In: *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, ACM, New York, NY, USA, pp. 91–100, doi:10.1145/1755952.1755967.

[6] Khaled El-Fakih, Nina Yevtushenko, Sergey Buffalov & Gregor v. Bochmann (2006): *Progressive solutions to a parallel automata equation*. *Theoretical Computer Science* 362(13), pp. 17 – 32, doi:10.1016/j.tcs.2006.05.034. Available at `http://www.sciencedirect.com/science/article/pii/S0304397506003161`.

[7] G.C Fox, S.W Otto & A.J.G Hey (1987): *Matrix algorithms on a hypercube I: Matrix multiplication*. *Parallel Computing* 4(1), pp. 17 – 31, doi:10.1016/0167-8191(87)90060-3. Available at `http://www.sciencedirect.com/science/article/pii/0167819187900603`.

[8] Wenzhong Gao & Shaobu Wang (2010): *On-line dynamic state estimation of power systems*. In: *North American Power Symposium (NAPS), 2010*, pp. 1 –6, doi:10.1109/NAPS.2010.5619951.

[9] I. Gorton, Zhenyu Huang, Yousu Chen, B. Kalahar, Shuangshuang Jin, D. Chavarria-Miranda, D. Baxter & J. Feo (2009): *A High-Performance Hybrid Computing Approach to Massive Contingency Analysis in the Power Grid*. In: *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, pp. 277 –283, doi:10.1109/e-Science.2009.46.

[10] Zhenyu Huang, Yousu Chen & J. Nieplocha (2009): *Massive contingency analysis with high performance computing*. In: *Power Energy Society General Meeting, 2009. PES '09. IEEE*, pp. 1 –8, doi:10.1109/PES.2009.5275421.

[11] Peter Hui, Satish Chikkagoudar & Daniel Chavarría-Miranda (2011): *Towards a Real-Time Cluster Computing Infrastructure*. *IEEE Real-Time Systems Symposium (Work-in-Progress Session)*. Available at `http://www.cs.wayne.edu/~fishern/Meetings/wip-rtss2011/`.

[12] Peter Hui, Barry Lee & Satish Chikkagoudar (2012): *Towards Real-Time High Performance Computing for Power Grid Analysis*. In: *Proceedings of the Second International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*, HiPCNA-PG '12, IEEE, Washington, DC, USA. To appear.

[13] M.J. Koop, T. Jones & D.K. Panda (2008): *MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand*. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1 –12, doi:10.1109/IPDPS.2008.4536283.

[14] P. David Stotts & William Pugh (1994): *Parallel finite automata for modeling concurrent software systems*. *J. Syst. Softw.* 27(1), pp. 27–43, doi:10.1016/0164-1212(94)90112-0.

[15] CORPORATE The MPI Forum (1993): *MPI: a message passing interface*. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, ACM, New York, NY, USA, pp. 878–883, doi:10.1145/169627.169855.