# Timed Signalling Processes

Rajit Manohar*
*Yale University*
*New Haven, CT 06520, USA*
rajit.manohar@yale.edu

Yoram Moses†
*Technion-Israel Institute of Technology*
*Haifa 32000, Israel*
moses@ee.technion.ac.il

***Abstract*—Circuits often use knowledge of time to order actions in a computation. The commonly used bundling constraint in bundled-data circuits states that a request signal must arrive only after the corresponding data wires have the correct value. Various informal and formal mechanisms have been used by designers to capture sufficient conditions for such constraints to be satisfied, including relative timing, pulse width requirements, and regions where signal changes are prohibited.**

**We study the problem of ordering signal transitions in an asynchronous computation when there is knowledge of wire delay and computation delay, but where time is not avaiable directly as a variable to any participating process. In this context, we introduce two signalling patterns: a timing fork, and a novel structure we call a zigzag pattern. We show that a zigzag pattern is sufficient to order signal transitions in the timed asynchronous setting. More importantly we show that if two signal transitions are ordered, then there exists a generalized zigzag pattern that guarantees their ordering.**

**This shows that a zigzag pattern is the fundamental construct needed to order signal transitions in the timed asynchronous circuit context. We show how such patterns capture commonly used timing constraints in practical asynchronous circuits.**

***Keywords*—*timing, event ordering, asynchronous circuits***

## I. Introduction

The delays of gates and wires in a circuit can impact both the performance of an asynchronous computation as well as its functionality. Apart from purely delay-insensitive circuits where the computation performed does not depend on the delays of gates and wires, all other asynchronous circuit families rely on either implicit or explicit assumptions about the delays of gates and/or wires in the physical realization of the circuit to ensure correct operation.

There are many ways that a designer specifies these delay requirements. Examples of constraints include: a path through a sequence of gates must have a delay larger than a specified value; a signal must have a minimum pulse width; and timing constraints that require that a particular signal transition must occur before another one. Researchers have developed a range of asynchronous circuit families, each of which make a different set of assumptions about timing. Examples of such families include delay-insensitive logic, quasi delay-insensitive logic [1], micropipelines [2], MOUSETRAP pipelines [3], GasP logic [4], and general timed circuits [5].

In this paper, we study the problem of ensuring that signal transitions are ordered in an asynchronous computation. The "pure" asynchronous setting where we make no assumptions about the delays of gates or wires is captured by recent work on asynchronous signalling processes [6]. In that setting, the only way two signal transitions can be ordered is if one is in the past of the other, and this can only be achieved if the circuit has an explicit path from one signal to the other [6].

In this paper, we assume that a designer has knowledge of delay bounds on wires and gates and can use this information to order signal transitions. We introduce the *timed signalling processes* model for an asynchronous system consisting of a collection of concurrent processes interacting via wires. We explictly model timing information regarding both the durations of the internal operations of computing elements (*processes*) and of communication between processes. In principle, the proposed framework is quite general, allowing a process to be anything from a simple Boolean gate to a much larger system. However, our immediate goal is to facilitate reasoning about timing in circuits. Consequently, we assume that communication between processes occurs over wires that communicate Boolean values of 0 or 1. Our setting assumes the existence of upper and lower bounds on the time it takes for a process to change its state, and bounds on the time it takes for a wire to communicate a value.

Any asynchronous circuit can be described using our model. In fact, our model can also describe synchronous circuits as a special case. Moreover, our setting makes it convenient to model circuits where different modules are individually governed by different timing schemes, which can easily occur in a large-scale asynchronous system.

In the timed signalling processes model, we introduce two communication structures: a *timing fork*, and a *zigzag pattern* that consists of a collection of timing forks in a specific configuration.[1] We prove two fundamental results on signal ordering: (i) a zigzag pattern ensures that two signal transitions are always ordered in a computation; and (ii) if two signal transitions are always ordered in any computation, then a generalized zigzag pattern must exist. In particular, either a zigzag pattern exists, or the model admits a computation that is potentially erroneous. Our results rigorously establish that a zigzag pattern is both necessary and sufficient to order events in the timed signalling processes model. Finally, we provide a number of examples that illustrate how common timing constraints are expressed using timing forks, including those required for flip-flop based synchronous logic.

† Yoram Moses is the Israel Pollak academic chair at the Technion, and was supported in part by the BSF grant 2015820 which is coincident with NSF-BSF grant CCF 1617945.
* Rajit Manohar was supported in part by DARPA IDEA grant FA8650-18-2-7850, and in part by DARPA POSH grant HR001117S0054-FP-042.

[1]Zigzag patterns are inspired by a similar communication structure in synchronous message-passing distributed systems introduced in [7].

## II. Definitions and Model

### A. Wires

The basic element used for communication between processes is a wire. A wire $\mathsf{w}$ consists of a pair of variables $(o_\mathsf{w}, i_\mathsf{w})$ in which $o_\mathsf{w}$ is called the wire's *originating variable* and $i_\mathsf{w}$ its *destination variable*. The variables $o_\mathsf{w}$ and $i_\mathsf{w}$ are considered *matching* variables. The former serves as an output of a process, and the latter is an input that the wire feeds into another process. The possible values of a destination variable are simply $\{0, 1, ?\}$. The possible values of an originating variable are $\{0, 1, ?, \mathsf{Z}\}$, where $\mathsf{Z}$ indicates that the originating variable is no longer being actively set to a value. The value ? indicates that the variable is in an unknown state.

*Definition 1 (enabled to signal):* When the value of the originating variable $o_\mathsf{w}$ is 0 and that of $i_\mathsf{w}$ is not 0, we say that wire $\mathsf{w}$ is *enabled to signal* 0. Similarly, if $o_\mathsf{w}$ is 1 (resp. ?) and $i_\mathsf{w}$ is not 1 (resp. ?), we say that $\mathsf{w}$ is *enabled to signal* 1 *(resp. ?).*

*Definition 2 (signalling):* A signal is delivered over $\mathsf{w}$ in a transition that starts with $\mathsf{w}$ being enabled to signal, and ends with the value of its destination variable $i_\mathsf{w}$ assigned to the value that $o_\mathsf{w}$ had initially (while $o_\mathsf{w}$ remains unchanged).

When $o_\mathsf{w}$ has value $\mathsf{Z}$, the wire $\mathsf{w}$ is disabled and no change in the destination variable can take place.[2]

Our goal is to explicitly model time bounds for signal propagation in wires. To this end, we associate two integer values with each wire $\mathsf{w}$: (i) $\mathsf{lb}(\mathsf{w}) \geq 1$, a lower bound on the wire delay, and (ii) $\mathsf{ub}(\mathsf{w}) \geq \mathsf{lb}(\mathsf{w})$, an upper bound on the wire delay. In addition, we permit $\mathsf{ub}(\mathsf{w})$ to be $\infty$ to model wires that can have arbitrarily large delay.

Given a set of originating variables $O$, we denote by $S^O$ the set of possible assignments $\sigma_O \colon O \to \{0, 1, ?, \mathsf{Z}\}$. Similarly, given a set of destination variables $I$, we denote by $S^I$ the set of possible assignments $\sigma_I \colon I \to \{0, 1, ?\}$.

### B. Processes

A *process* in our framework is associated with a set $\mathsf{Q}$ of internal states. In addition, each process $p$ is associated with a set $O = O^p$ of originating variables called its outputs, and a set $I = I^p$ of destination variables called its inputs. A *local state* of such a process consists of a triple $\ell = (\mathsf{q}, \sigma_O, \sigma_I)$, where $\mathsf{q} \in \mathsf{Q}$ is the internal state, $\sigma_O \colon O \to \{0, 1, ?, \mathsf{Z}\}$ maps output variables to their values, and $\sigma_I \colon I \to \{0, 1, ?\}$ maps input variables to their values. The actions of a process can change its internal state $\mathsf{q}$ and output variables $\sigma_O$. We call the pair $(\mathsf{q}, \sigma_O)$ the *controllable* (local) state of the process.

Formally, a *process* in our framework is a tuple $p = (\mathsf{Q}, \hat{\delta}, O, I)$ that describes a nondeterministic automaton where $\mathsf{Q}$, $O$, and $I$ are as above with the special state $\perp \in \mathsf{Q}$ (and hence $|\mathsf{Q}| \geq 2$), and $\hat{\delta} \colon \mathsf{Q} \times S^O \times S^I \to (2^{\mathsf{Q} \times S^O} \setminus \emptyset)$ is its nondeterministic transition function. The special state $\perp$ is used to indicate that the process is in an erroneous state. Intuitively, $\hat{\delta}$ sets the next controllable state of a process $p$ based on $p$'s current local state.[3] A process is said to be *quiescent* in a local state $(\mathsf{q}, \sigma_O, \sigma_I)$ if $\hat{\delta}(\mathsf{q}, \sigma_O, \sigma_I) = \{(\mathsf{q}, \sigma_O)\}$. If a process is quiescent, then one of its inputs must change before its controllable state can change. A *step* of the process in local state $(\mathsf{q}, \sigma_O, \sigma_I)$ changes its controllable state from $(\mathsf{q}, \sigma_O)$ to $(\mathsf{q}', \sigma'_O)$ where $(\mathsf{q}', \sigma'_O) \in \hat{\delta}(\mathsf{q}, \sigma_O, \sigma_I)$. In general, different outputs of a process can have distinct values.

As we shall see in Section II-D, there may be situations where there is a conflict in potential next states that can lead to erroneous behavior. To capture this, we define the *unification* operator between two possible sets of next states. If $S_1$ and $S_2$ are contained in $2^{\mathsf{Q} \times S^O}$, then $S_1 \uplus S_2$ is defined to be $S_1$ if $S_1 = S_2$; otherwise it is defined to be $\{(\perp, \sigma^*_O)\}$, where $\sigma^*_O$ maps an output variable to 0 (resp. or 1 or $\mathsf{Z}$) iff all possible $\sigma_O$ assignments in $S_1 \cup S_2$ map the output variable to the same value; otherwise, the output variable is mapped to ?.

In the timed setting, processes take some time to process an input change. We associate a lower bound $\mathsf{lb}(p) > 0$ and an upper bound $\mathsf{ub}(p) \geq \mathsf{lb}(p)$ with each process $p$ that specify how long the process $p$ might take to change its controllable state. We permit $\mathsf{ub}(p)$ to be $\infty$. For this delay to reflect the time it takes for the process to take one step, we require that whenever a process takes a step, it enters a quiescent state—i.e., that for all $(\mathsf{q}', \sigma'_O) \in \hat{\delta}(\mathsf{q}, \sigma_O, \sigma_I)$, we have $\hat{\delta}(\mathsf{q}', \sigma'_O, \sigma_I) = \{(\mathsf{q}', \sigma'_O)\}$.

Our goal is to analyze how timing information can be used to reason about computations using lower and upper bounds on signal propagation and on process steps. If a process could take multiple steps (through internal state evolution) even when its inputs and outputs remain unchanged, then the delay taken for a process to change its output would be state-dependent. The requirement that a process enter a quiescent state after a step eliminates this possibility, and provides sufficient detail in our model so that we can analyze timing properties by examining upper and lower bounds. Note that in the simplest case where a process is an individual gate, the quiescent state requirement is guaranteed to be true, so this constraint does not limit the class of circuits that can be described using our model.

Given a set of processes $P$, a *configuration* $c$ of $P$ assigns a local state $c(p) = (\mathsf{q}, \sigma_O, \sigma_I)$ and an integer $\mathsf{k}_p < \mathsf{ub}(p)$ to every process $p \in P$, and assigns an integer $\mathsf{k}_\mathsf{w}$ to each wire $\mathsf{w}$. For a wire that is enabled to signal, $\mathsf{k}_\mathsf{w}$ tracks how long it has been enabled. For a process $p$, the integer $\mathsf{k}_p$ tracks how long it has been since the process last exited a quiescent state.

If $p$'s local state in a configuration $c$ is $c(p) = (\mathsf{q}, \sigma_O, \sigma_I)$, then we denote the controllable state of $p$ by $c^\dagger(p) \overset{\text{def}}{=} (\mathsf{q}, \sigma_O)$.

### C. Networks of Timed Signalling Processes

A *TSP network* is a triple $\mathcal{S} = (P, W, C_0)$ consisting of a set of processes $P$, a set $W$ of the wires appearing in $P$, and a set $C_0$ of configurations of $P$. This set $C_0$ is called the set of *initial configurations*, and will determine the possible initial states of the system. To ensure that $W$ represents the input/output variables in $P$, we require:

[2]We have chosen not to have the originating variable value $\mathsf{Z}$ carry the information of whether the last value "delivered" on the wire was 0 or 1. In our formalism, this information can be captured by the internal state of the process where the wire originates, if needed.

[3]Since we allow the transition function to be nondeterministic, our processes can correspond to a rich variety of devices, including synchronous and asynchronous gates or sub-circuits.

1) Every input and output variable in a process $p \in P$ belongs to a wire in $W$, and
2) For every wire $\mathsf{w} = (o_\mathsf{w}, i_\mathsf{w}) \in W$ there are two unique processes $p, p' \in P$ such that $o_\mathsf{w}$ is an originating variable (output) of $p$ and $i_\mathsf{w}$ is a destination variable (input) of $p'$.

In order to model circuits by a TSP, we will introduce two special processes to capture initialization; these are detailed in Section III-C. (The definitions that follow do not depend on the specific nature of these two processes.)

It is sometimes convenient to consider the communication graph $G_\mathcal{S} = (V_\mathcal{S}, E_\mathcal{S})$ underlying a given TSP network $\mathcal{S} \overset{\text{def}}{=} (P, W, C_0)$. This is a directed graph with vertex set $V_\mathcal{S} = P$ consisting of the processes of $\mathcal{S}$, and where there is a directed edge $(q, p) \in E_\mathcal{S}$ iff there are matching output and input variables $o_\mathsf{w} \in O^q$ and $i_\mathsf{w} \in I^p$. The associated wire $\mathsf{w}$ is considered an *output wire* of $q$ and an *input wire* of $p$. Note that while a process typically has external inputs and outputs, a TSP network has neither. In this sense, a process may be open to external interaction, but a TSP system is closed—i.e., it includes the circuit being modeled as well as the environment. Since we have complete knowledge of the entire system, a "full system" analysis might provide insights into timed behavior that would not be available when a collection of processes is analyzed in isolation.

## D. Computations and Valid Computations

A *computation* of a TSP network $\mathcal{S} = (P, W, C_0)$ is a sequence $c = c_0, c_1, c_2, \ldots$ of configurations of $\mathcal{S}$. A *valid computation* of a TSP network $\mathcal{S} = (P, W, C_0)$ is a computation in which $c_0 \in C_0$ is an initial configuration of $\mathcal{S}$, and for all $j \geq 0$, configuration $c_{j+1}$ is obtained from $c_j$ by applying the following sets of rules to processes and wires.

*Processes:* The following rules apply to determine $c^\dagger_{j+1}(p)$ from $c_j(p)$. If more than one rule applies, the earlier rule listed is given precedence.

- **(idle)** If a process $p$ is quiescient at time $j$, then $\mathsf{k}_p$ is set to 0 at time $j + 1$ (i.e. in $c_{j+1}$) and $c^\dagger_{j+1}(p) = c^\dagger_j(p)$;
- **(min-delay)** if $\mathsf{k}_p + 1 < \mathsf{lb}(p)$ then $\mathsf{k}_p$ is incremented by one, and $c^\dagger_{j+1}(p) = c^\dagger_j(p)$;
- **(act/delay)** if $\mathsf{lb}(p) \leq \mathsf{k}_p + 1 < \mathsf{ub}(p)$, then either
  - **(delay)** $\mathsf{k}_p$ is incremented by one and $c^\dagger_{j+1}(p) = c^\dagger_j(p)$, or
  - **(act)** $\mathsf{k}_p$ is set to 0 and process $p$ takes a step—which corresponds to picking $(\mathsf{q}', \sigma'_{O^p}) \in \hat{\delta}(c_j(p))$ and updating its controllable state from $c^\dagger_j(p) = (\mathsf{q}, \sigma_{O^p})$ to $c^\dagger_{j+1}(p) = (\mathsf{q}', \sigma'_{O^p})$;
- **(max-act)** if $\mathsf{k}_p + 1 = \mathsf{ub}(p)$ then process $p$ takes a step (as in the **act** rule above) and $\mathsf{k}_p$ is set to 0.

*Wires:* For each wire $\mathsf{w} = (o_\mathsf{w}, i_\mathsf{w})$:

- **(idle)** if $\mathsf{w}$ is not enabled to signal in $c_j$, then $\mathsf{k}_\mathsf{w}$ is set to 0 and $i_\mathsf{w}$ is unchanged in $c_{j+1}$;
- **(min-delay)** if $\mathsf{w}$ is enabled to signal in $c_j$ and $\mathsf{k}_\mathsf{w} + 1 < \mathsf{lb}(\mathsf{w})$, then $\mathsf{k}_\mathsf{w}$ is incremented by one and $i_\mathsf{w}$ is unchanged in $c_{j+1}$;
- **(act/delay)** if $\mathsf{w}$ is enabled to signal in $c_j$ and $\mathsf{lb}(\mathsf{w}) \leq \mathsf{k}_\mathsf{w} + 1 < \mathsf{ub}(\mathsf{w})$, then either
  - **(delay)** $i_\mathsf{w}$ is unchanged and $\mathsf{k}_\mathsf{w}$ is incremented by one in $c_{j+1}$, or
  - **(act)** a signal is delivered over $\mathsf{w}$ (i.e., $i_\mathsf{w}$ is set to the value of $o_\mathsf{w}$) and $\mathsf{k}_\mathsf{w}$ is set to 0 in $c_{j+1}$;
- **(max-act)** if $\mathsf{w}$ is enabled to signal in $c_j$ and $\mathsf{k}_\mathsf{w} + 1 = \mathsf{ub}(\mathsf{w})$, a signal is delivered over $\mathsf{w}$ (i.e., $i_\mathsf{w}$ is set to $o_\mathsf{w}$) and $\mathsf{k}_\mathsf{w}$ is set to 0 in $c_{j+1}$.

If $i_\mathsf{w}$ in $c_{j+1}$ differs from $i_\mathsf{w}$ in $c_j$, then we say that a *signal is delivered* over wire $\mathsf{w}$ at time $(j + 1)$ in computation $c$.

We also apply the following rule after the ones above:

- **(reset)** If $\mathsf{k}_p$ is set to a non-zero value in $c_{j+1}$ by the rules above and $\hat{\delta}(c_{j+1}(p)) \neq \hat{\delta}(c_j(p))$, then $\mathsf{k}_p$ is set to 0 in $c_{j+1}$ and the controllable state is set to $\hat{\delta}(c_j(p)) \uplus \hat{\delta}(c_{j+1}(p))$.

If $\mathsf{k}_p$ is set to a non-zero value in $c_{j+1}$ by the rules above and $\hat{\delta}(c_{j+1}(p)) = \hat{\delta}(c_j(p))$ but $c_{j+1}(p) \neq c_j(p)$, we say that $p$ *absorbs* the input change at time $j$. Note that $c_{j+1}(p) \neq c_j(p)$ is another way of stating that some input of $p$ changed, because $c^\dagger_{j+1}(p) = c^\dagger_j(p)$ when $\mathsf{k}_p > 0$.

The process rules determine the new values of $\mathsf{q}$, $\sigma_{O^p}$, and $\mathsf{k}_p$ for each process $p$. The wire rules determine the new values of $\sigma_{I^p}$ for each process $p$, and the new values of $\mathsf{k}_\mathsf{w}$ for each wire $\mathsf{w}$. Finally, the **reset** rule is applied to reset the process delay counter if an input change causes the possible next states to be modified.

For a computation $s = c_0, c_1, c_2, \ldots$, we use $s(t)$ to denote the configuration $c_t$, and we use $s_p(t)$ to denote the local state $(\mathsf{q}, \sigma_O, \sigma_I)$ of process $p$ in configuration $c_t$.

The definition of a valid computation captures the following intuitions about signal propagation over bounded delay wires. If a wire is enabled to signal for a time period smaller than its minimum propagation delay, it cannot signal (rule **min-delay**); if the period reaches the upper bound of the propagation delay, then it must signal (rule **max-act**); in the intermediate cases, it may signal and the choice is non-deterministic (rule **act/delay**).

Recall that a signal that is delivered at time $t$ is reflected in the input variable of the receiving process at time $t$; hence, the step taken by the process at time $t$ can, in general, depend on the signal's updated value. This is the convention we adopt when we use the phrase "delivered at time $t$."

Note that while process steps have bounded delays in our model, the timer $\mathsf{k}_p$ that we use in the model to track the delay is not part of any local state—and is therefore not accessible to the processes.

## E. Erroneous States and ?-Signalling

Up to this point the state $\perp$ and the signal value ? have been treated just as an ordinary state and signal. We now introduce new rules and constraints that capture the intuition that these two symbols are meant to refer to erroneous behavior and switching hazards.

If $\mathsf{w}$ is enabled to signal in $c_j$, a signal is not delivered over $\mathsf{w}$ at time $(j + 1)$, and $o_\mathsf{w}$ at time $(j + 1)$ differs from $o_\mathsf{w}$ at time $j$, then $i_\mathsf{w}$ is set to ? in $c_{j+1}$ and $\mathsf{k}_\mathsf{w}$ is set to 0. This models a switching hazard on $\mathsf{w}$, and we say that the signal delivery was *cancelled*.

To capture the fact that a "?" input cannot be used to provide information, we impose the following constraint on $\hat{\delta}$. We say that $\sigma' \approx \sigma$ if the two functions agree on all non-? values.

$$\hat{\delta}(\mathsf{q}, \sigma_I, \sigma_O) = \biguplus_{\sigma'_I \approx \sigma_I, \sigma'_O \approx \sigma_O} \hat{\delta}(\mathsf{q}, \sigma'_I, \sigma'_O)$$

Note that the value of $\hat{\delta}$ when an input or output variable is "?" can be determined from its value when all inputs and outputs are non-"?" values. We also impose the constraint that an output variable of a process can be "?" state if and only if the process is in the "$\perp$" state.

Finally, the constraint we impose on the $\perp$ state is a process in the $\perp$ state is either quiescient, or any non-quiescient transition must take it out of the $\perp$ state.

### F. Self-timed Circuits as TSP Networks

There are many possible valid computations of a TSP network that start with a particular initial configuration $c_0$. The existence of lower and upper bounds for signal propagation and process execution permits computations to use timing information for coordinating activity.

TSP systems provide a unified setting to model many asynchronous circuit families. The class of purely delay-insensitive (DI) circuits can be directly captured by mapping gates to processes with a single state each, where the function $\hat{\delta}$ for a gate captures the behavior of the gate's pull-up and pull-down switching logic. All upper bounds are infinite, and all lower bounds are set to 1.

All other asynchronous circuit families introduce timing assumptions either on wires, on gates, or on both. Speed-independent circuits can be modeled by setting both the upper and lower bound on wire delays to 1, with processes (used to model gates) having a delay with a lower bound of 1 and upper bound of infinity. For circuit families that use more knowledge about gate and wire delays, assumptions about minimum and maximum delay of gates and wires can be encoded using the lower and upper bounds available in the TSP model.

*Example: bundled-data channels:* A bundled-data communication channel uses $n$ data wires together with a *request* and *acknowledge* wire to transmit $n$ bits of information. In the simplest version of the channel, the bundling constraint is that the data wires must be correctly set when the request wire transitions from zero to one, and the data can be modified once the acknowledge transitions from zero to one. Such a communication protocol cannot be handled by purely asynchronous signalling processes that permit arbitrarily large delays on every wire in the system. The TSP model can bound delays on circuit components and wires that are used to drive the bundled data channel signals, and hence can be used to establish whether or not the bundling constraint is satisfied.

### G. Spontaneous Events and Wire Fanout

It is useful to include special processes that are the source of *spontaneous* events—events where we have no control over when they occur. This corresponds, for example, to when an input signal might change at an unknown time to initiate part of a computation.

*Definition 3 (spontaneous process):* A process $p = (\mathsf{Q}, \hat{\delta}, O, I)$ in a TSP network is said to be a spontaneous 0-to-1 process iff it satisfies the following:

- $\mathsf{Q} = \{init\}$
- $I = \emptyset$ and $O = \{o_{\mathsf{w}}\}$ (a singleton)
- In every initial configuration of the TSP network, $o_{\mathsf{w}} = 0$ and the matching input variable $i_{\mathsf{w}} = 0$
- $\hat{\delta}(init, \sigma_I, \sigma_O) = \{(init, (o_{\mathsf{w}} \mapsto 1))\}$, where $(x \mapsto v)$ denotes a partial function that maps $x$ to $v$
- $\mathsf{ub}(p) = \infty$ and $\mathsf{lb}(p) = 1$

A spontaneous 1-to-0 process is similar, except the state transition function flips the output variable $o_{\mathsf{w}}$ from 1 to 0.

Our model only has point-to-point wires. To support wire fanout, we introduce a special process called a *deterministic wire fork*.

*Definition 4 (deterministic wire fork):* A deterministic wire fork is a process with delay bounds of $[1, 1]$, a single input, and multiple output variables with output wire delay bounds of $[1, 1]$. The wire fork simply propagates any input change to its set of outputs with a fixed delay.

## III. Timed Signalling Processes

The sequence defined by a computation is indexed by an integer, which plays the role of an external notion of time. (We use both $t$'s and $m$'s for these indexes.) But, in the setting we consider, processes in the system do not have access to the current time, and it does not affect their operation. However, processes can learn about the passage of time through input changes coupled with guarantees about time bounds.

### A. Timed Precedence

In what follows, we use notation borrowed from [8] that states timed precedence relations between events. We write $e \xrightarrow{x} e'$ to state that event $e$ takes place at least $x$ time units before $e'$ does. This orders events $e$ and $e'$, and also places a lower bound on the time difference between events $e$ and $e'$. A direct example of such timed precedence in asynchronous circuits is in the case of bundled data communication discussed earlier. The value of $x$ corresponds to the timing margin in the timing assumption.

Suppose we wish to ensure that $b \xrightarrow{0} c$ where $b$ is an action of $B$ and $c$ is an action of $C$. A direct way to ensure this would be for process $B$ to have an output wire that connects to one of $C$'s inputs. Once $B$ has performed action $b$, then it could signal this to $C$ by changing the output.

Due to known time bounds on signal propagation, such a timed relationship can also be guaranteed *indirectly* via a third process $A$. Suppose $A$ has an output that is connected to $B$'s input, and another output connected to $C$'s input. If the delay from $A$ to $B$ is smaller than that from $A$ to $C$, and both $B$ and $C$ act in response to a signal from $A$, then knowledge of time bounds on signal propagation and on processing delays may be sufficient to ensure that $b \xrightarrow{0} c$.

## *B. Nodes*

To reason about such relationships and to distinguish the information available to a process at different times, it is useful to be able to refer to a process when it is in a particular local state. Since the number of distinct local states of a process is finite, a process may leave a particular state at some point in time and then return to it at a later point. This fact is not observable by the process itself, but reasoning from the outside we would like to be able to distinguish between these different occurrences of the same state in a computation. To do so, we introduce the notion of an *occurrence index*.

*Definition 5 (occurrence index):* Given a computation $c$, process $p$, and integer $m$, we define $\mathbf{idx}(c,p,m)$ inductively as follows. Base case: $\mathbf{idx}(c,p,0)=0$ for all $c$ and $p$. For the inductive step, we use case analysis as follows:

(a) if $c_{k+1}(p)=c_k(p)$, then $\mathbf{idx}(c,p,k+1)=\mathbf{idx}(c,p,k)$;
(b) if $c_{k+1}(p)\neq c_m(p)$ for all $m\leq k$, then we define $\mathbf{idx}(c,p,k+1)=0$;
(c) otherwise define $\mathbf{idx}(c,p,k+1)=\mathbf{idx}(c,p,m)+1$ where $m<k$ is the largest integer such that $c_m(p)=c_{k+1}(p)$.

We define $\nu=\langle p,l,i\rangle$ consisting of process $p$ with its local state $l$ and an integer occurrence index $i\geq 0$ to be a *node*. We say that $\nu$ is a *$p$-node* as it refers to process $p$, and use $\mathtt{proc}(\nu)$ to denote $p$. Given a computation $c$, we say that node $\nu=\langle p,l,i\rangle$ *appears in* $c$ if $c_m(p)=l$ and $\mathbf{idx}(c,p,m)=i$ holds, for some time $m$.

If a node $\nu=\langle p,l,i\rangle$ appears in a computation $c$ at time $m$, the process $p$ may continue to be in local state $l$ for some interval of time that includes $m$. We define $\mathbf{time}_c(\nu)$ to be the smallest (i.e. earliest) $m$ such that $c_m(p)=l$ and $\mathbf{idx}(c,p,m)=i$.

A given computation is uniquely determined by the set of its nodes and their respective times, but different executions can lead to computations with the same set of nodes that differ in their timing.

A node $\nu'$ is called the *next local node* of a node $\nu$ in $c$ if $\mathtt{proc}(\nu)=\mathtt{proc}(\nu')$, $\mathbf{time}_c(\nu)<\mathbf{time}_c(\nu')$, and there is no other $\mathtt{proc}(\nu)$-node $\nu''$ such that $\mathbf{time}_c(\nu)<\mathbf{time}_c(\nu'')<\mathbf{time}_c(\nu')$.

We remark that the definitions above of nodes, **time** and **idx** can be applied to any computation, not just valid computations.

*Definition 6 (successor):* Let $p$ and $q$ be two processes connected by a wire $\mathsf{w}$ where $o_\mathsf{w}$ is an output variable of $p$ and $i_\mathsf{w}$ is its matching input variable in $q$. Suppose that $c$ is a computation where (i) $p$-node $\nu_p$ first appears at time $(t-1)$ and $o_\mathsf{w}$ at time $t$ differs from $o_\mathsf{w}$ at time $(t-1)$; and (ii) either (a) a signal from $p$ is delivered to $q$ over $\mathsf{w}$ at time $(t+d)$ where $d\geq 1$;[4] (b) $o_\mathsf{w}$ is unchanged for all times between $t$ and $(t+d)$; and (c) the $q$-node at time $(t+d)$ is $\nu_q$; or (a) $o_\mathsf{w}$ at time $t$ differs from $o_\mathsf{w}$ at time $(t-1)$ causing $i_\mathsf{w}$ to be set to ? at $t$ and $\nu_q$ is the $q$-node at time $t$. Then we write $\nu_p\hookrightarrow_c\nu_q$ and call $\nu_q$ a *successor* of $\nu_p$.

Intuitively, this captures the fact that a signal from $p$ changes an input variable in $q$, and can causally affect the future behavior of $q$. We capture the temporal ordering between two $p$-nodes as follows:

*Definition 7 (local ordering):* Suppose $\nu$ and $\nu'$ are both nodes for the same process $p$ in a computation $c$. If $\mathbf{time}_c(\nu)<\mathbf{time}_c(\nu')$, we write $\nu<^c_\ell\nu'$; similarly if $\mathbf{time}_c(\nu)\leq\mathbf{time}_c(\nu')$, we write $\nu\leq^c_\ell\nu'$. The *local distance* between $\nu$ and $\nu'$, denoted $\mathbf{dist}_c(\nu,\nu')$ is the number of distinct nodes $\nu''$ where $\nu<^c_\ell\nu''\leq^c_\ell\nu'$.

These two ordering relations can be combined to define a notion of "happens before" or potential causality as follows.

*Definition 8 (happens-before):* Given a computation $c$, we define the *happens-before* relationship among nodes that appear in $c$, denoted $\nu\prec_c\nu'$ to be the minimal relation that satisfies

- Locality: for nodes of the same process, if $\nu<^c_\ell\nu'$ then $\nu\prec_c\nu'$
- Signalling: if $\nu\hookrightarrow_c\nu'$, then $\nu\prec_c\nu'$
- Transititivity: if $\nu\prec_c\nu'$ and $\nu'\prec_c\nu''$, then $\nu\prec_c\nu''$

We say that $\nu'$ is *in the past* of $\nu$ in $c$ if $\nu'\prec_c\nu'$, and define $\mathbf{past}(c,\nu)=\{\nu':\nu'\prec_c\nu\}$. Nodes that are in the set $\mathbf{past}(c,\nu)$ either because they refer to the the process $\mathtt{proc}(\nu)$, or via a causal succession of signals over wires. We make the latter precise by introducing the notion of *signalling chains*.

*Definition 9:* A ***signalling chain from $\nu$ to $\nu'$ in the computation*** $c$ is a sequence of nodes $\langle\mu_0,\nu_1,\mu_1,\ldots,\nu_k\rangle$ with $\mu_0=\nu$, $\nu_k=\nu'$, $\mu_i\hookrightarrow_c\nu_{i+1}$ for $0\leq i<k$, and with $\nu_j\leq^c_\ell\mu_j$ for $1\leq j<k$.

The notion of a signalling chain is analogous to a message chain in a distributed computation [9], to a firing chain in asynchronous quasi delay-insensitive circuits [10], and to a signalling chain in the pure asynchronous signalling process model [6]. Associated with each signalling chain is a path in the TSP network's communication graph.

The fact that '$\prec_c$' is the minimal relation satisfying the Locality, Successor and Transitivity conditions means that $\nu\prec_c\nu'$ holds only if it can be derived by a finite number of applications of these conditions. Based on this, a rather straightforward consequence of the definition of $\prec_c$ is the following lemma:

*Lemma 1:* For nodes $\nu=\langle p,l,i\rangle$ and $\nu'=\langle q,l',i'\rangle$ where processes $p\neq q$, if $\nu\prec_c\nu'$ then $\mathbf{time}_c(\nu)<\mathbf{time}_c(\nu')$, and there exists a signalling chain from $\mu$ to $\mu'$ in $c$ where nodes $\mu$ and $\mu'$ satisfy $\nu\leq^c_\ell\mu$ and $\mu'\leq^c_\ell\nu'$.

**Proof:** The proof can be constructed by structural induction on the derivation of $\nu\prec_c\nu'$ using the rules for locality, signalling, and transitivity. □

Lemma 1 immediately implies that if $\nu\prec_c\nu'$ then there is a path from $\mathtt{proc}(\nu)$ to $\mathtt{proc}(\nu')$ in the TSP network graph.

Consider a process $A$ that has outputs connected to processes $B$ and $C$ via wires $\mathsf{w}_B$ and $\mathsf{w}_C$ respectively, and suppose that $A$ changes its output to $B$ and $C$ simultaneously. Then, the input to $B$ will change $t$ time units later, where $\mathsf{lb}(\mathsf{w}_B)\leq t\leq\mathsf{ub}(\mathsf{w}_B)$, and $C$'s input will change $t'$ units later where $\mathsf{lb}(\mathsf{w}_C)\leq t'\leq\mathsf{ub}(\mathsf{w}_C)$. Notice that if $\mathsf{ub}(\mathsf{w}_C)<\mathsf{lb}(\mathsf{w}_B)$, then the arrival of a new signal from $A$ on $B$'s input indicates to $B$ that $C$ has already received a new signal on its input from $A$. Figure 1 illustrates this scenario.

[4]As stated earlier, a signal delivered at time $(t+d)$ is reflected in $i_\mathsf{w}$ at time $(t+d)$. Recall that $i_\mathsf{w}$ is part of $q$'s local state.

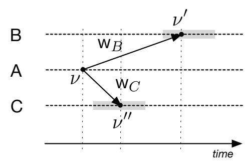

Fig. 1. $A$, $B$, and $C$ are signalling processes with two outputs from $A$ directly connected to $B$ and $C$ respectively. The gray region shows the time window when the signal could be delivered at $B$ and $C$ based on time bounds on wire delay. Nodes are shown on timelines of $A$, $B$, and $C$.

A signalling chain corresponds to a path through the TSP network graph, which consists of a sequence of processes and wires connecting neighboring processes. When an originating variable for a wire $\mathsf{w}$ changes, this signal is delivered over $\mathsf{w}$ after a delay that is consistent with the wire delay bounds $\mathsf{lb}(\mathsf{w})$ and $\mathsf{ub}(\mathsf{w})$. When a new input arrives taking a process $p$ out of a quiescient state, the process changes its controllable state at a time that is consistent with its delay lower bound $\mathsf{lb}(p)$ and delay upper bound $\mathsf{ub}(p)$ provided all other arriving inputs are absorbed until the process changes its controllable state. In this scenario, we have upper and lower bounds on the time taken by the process $p$ determined by $\mathsf{lb}(p)$ and $\mathsf{ub}(p)$.

A signalling chain can also include local state changes within an individual process—i.e., $\mu_i$ need not be the same as $\nu_i$. In this case, there may be other actions taken by $p_i = \mathtt{proc}(\nu_i)$, or other input arrivals at $p_i$ before the node $\nu_{i+1}$ appears in the computation. These input arrivals could result in the (**reset**) rule being applied, which can reset the process timer $\mathsf{k}_{p_i}$. Also, the input arrival corresponding to $\nu_i$ itself might be *absorbed* by $p_i$. In scenarios such as the ones just described, the the time taken by $p_i$ to act can depend on events in other processes, and not just on the local bounds $\mathsf{lb}(p_i)$ and $\mathsf{ub}(p_i)$. In the circumstances when the local process lower and upper bounds can be used to determine delay bounds and processes don't enter an erroneous state, we call the signalling chain a *signal delay chain*.

*Definition 10 (signal delay chain):* Let $\langle \mu_0, \ldots, \nu_k \rangle$ be a signalling chain from $\mu_0$ to $\nu_k$ in computation $c$. We say that this signalling chain is a *signal delay chain* if and only if the following constraints are satisfied: (i) all nodes in the chain are in a non-$\perp$ state; (ii) for all $1 \le i < k$, any input that arrives at $\mathtt{proc}(\nu_i)$ from time $\mathbf{time}_c(\nu_i)+1$ upto $\mathbf{time}_c(\mu_i)$ is absorbed; and (iii) for all $1 \le i < k$, $\mathsf{k}_{\mathtt{proc}(\nu_i)} = 0$ by a signal delivered by $\mathtt{proc}(\mu_{i-1})$. We write $\mu_0 \rightsquigarrow_c \nu_k$ to state that the computation $c$ contains a signal delay chain from $\mu_0$ to $\nu_k$.

Suppose we are given two nodes $\nu$ and $\nu'$ in a computation $c$ where $\nu \prec_c \nu'$. By Lemma 1, there is a signalling chain through a sequence of nodes $\mu_0, \nu_1, \mu_1, \ldots, \nu_k$ where $\nu \le_\ell^c \mu_0$ and $\nu_k \le_\ell^c \nu'$. Furthermore, $\mathtt{proc}(\nu_i) \ne \mathtt{proc}(\nu_{i+1})$ for all $0 \le i < k$. Let $\mathsf{w}_i$ be the wire connecting $\mathtt{proc}(\nu_{i-1})$ to $\mathtt{proc}(\nu_i)$ that is used by the signal delay chain. We define the signal chain bounds for chain $\rho = \mu_0 \rightsquigarrow_c \nu_k$ as:

$$\mathsf{lb}(\rho) = \mathsf{lb}(\mathsf{w}_k) + \sum_{i=1}^{k-1} (\mathsf{lb}(\mathsf{w}_i) + \max(\mathsf{lb}(\mathtt{proc}(\nu_i)), \mathbf{dist}_c(\nu_i, \mu_i)))$$

$$\mathsf{ub}(\rho) = \mathsf{ub}(\mathsf{w}_k) + \sum_{i=1}^{k-1} (\mathsf{ub}(\mathsf{w}_i) + \mathsf{ub}(\mathtt{proc}(\nu_i)))$$

These upper and lower bounds apply to the delay associated with the signalling chain if it is a signal delay chain. Note that if there are a sufficient number of local nodes between $\nu_i$ and $\mu_i$ (captured by $\mathbf{dist}_c(\nu_i, \mu_i)$), then that provides another lower bound on the number of steps before a signal is transmitted by process $\mathtt{proc}(\nu_i)$—hence we use the maximum between this quantity and the process delay lower bound.

Consider signal delay chains in a given computation from process $A$ to process $B$ and from $A$ to $C$ that share the starting $A$-node. Let $d_{A\to B}$ denote the delay along the signal delay chain $\rho$ from $A$ to $B$, and let $d_{A\to C}$ be the delay along the signal delay chain $\rho'$ from $A$ to $C$. To be able to unequivocally state that $d_{A\to C} < d_{A\to B}$, the maximum value of $d_{A\to C}$ must be smaller than the smallest value of $d_{A\to B}$. Given our time bounds, the smallest value of $d_{A\to B}$ is given by $\mathsf{lb}(\rho)$. Similarly, the largest value for $d_{A\to C}$ is given by $\mathsf{ub}(\rho')$.

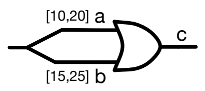

Fig. 2. A signal goes through a fork and arrives at the two inputs $a$ and $b$ of an OR gate with output $c$. The delay along the path to $a$ is given by the lower and upper bounds $[10, 20]$, and the other path has delay $[15, 25]$. Furthermore, let the $OR$ gate have a delay of $[1, 1]$.

In the general case, determining whether $d_{A\to C} < d_{A\to B}$ requires a complete analysis of the computations that a TSP system performs. To see this, consider the circuit example in Figure 2. Assume that all signals are initially 0, and the input changes to 1. This change will propagate via two wires with differing delays, and cause c to eventually become 1. Note that if a changes before b, then a causes c to change and there is a signal delay chain along the wire for a; furthermore, the change in b is absorbed. While there is a signalling chain that includes the wire for b to c, it is not a signal delay chain. The situation can be reversed in the symmetric case.

## C. Initialization and Initially Active Processes

Consider a computation $c$. Given an output signal transition that results in a new node $\nu$, we can find the last input change to $\mathtt{proc}(\nu)$ in $c$ that determined $\mathbf{time}_c(\nu)$. This input change in turn was the result of a previous output change at some node $\nu_{-1}$, whose time in turn was determined by an input change at $\mathtt{proc}(\nu_{-1})$. Repeating this process backward in time results in an initial node $\nu'$ and a signal delay chain $\nu' \rightsquigarrow_c \nu$. The time delay between $\nu'$ and $\nu$ in $c$ satisfies the bounds as defined above. Furthermore, $\nu'$ must change its output simply based on the initial state of the computation, and hence the time when this occurs is determined entirely by $\mathsf{lb}(\mathtt{proc}(\nu'))$ and $\mathsf{ub}(\mathtt{proc}(\nu'))$; therefore, in a precise sense, $\mathtt{proc}(\nu')$ is *initially active*.

Initially active processes are implicitly synchronized in time by the definition of a computation, since they all start at time $t = 0$. To allow processes to start operating at arbitrary times (not just start at $t = 0$), we introduce a dummy process that can delay the start time of a computation by an arbitrary amount.

To capture this in our model, we make use of two processes: the first is a spontaneous 0-to-1 process, and the second is a deterministic wire fork. The output of the spontaneous process is connected to the input of the wire fork, and each wire fork output is connected to a distinct initially active process.

Finally, the state transition function of each initially active process is modified to be quiescent when the newly introduced input is 0, and to behave in the same way as the original computation when the newly introduced input is 1. Note that (ignoring the newly introduced processes and wires) the resulting computations are precisely the same as we had prior to making this modification, apart from a uniform temporal shift that varies depending on the delay of the spontaneous process. Hence, this modification does not change the ordering and relative timing properties of the nodes in any computation. In the circuit context, this synchronization is typically achieved via a system-wide reset protocol. Our construction guarantees that there is a single spontaneous 0-to-1 process that is initially active, and the processes that would be initially active in its absence are now activated simultaneously.

We refer to TSP networks with this modification as *circuit TSP networks* (denoted cTSP), and these are the networks we consider in the remainder of this paper.

In the simple case when a signalling chain from $\nu$ to $\nu'$ consists of only one wire (as in the example in Figure 1), then we can state that $\nu \rightsquigarrow_c \nu'$ for every computation $c$ that includes a signalling chain from $\nu$ to $\nu'$.

## IV. Forks and Zigzags

Given two nodes $\theta$ and $\theta'$ in a computation $c$, we say that the computation $c$ satisfies $\theta \xrightarrow{x} \theta'$ iff (i) $\theta$ and $\theta'$ appear in $c$, and (ii) $\mathbf{time}_c(\theta) + x \leq \mathbf{time}_c(\theta')$.

The basic building block for timed coordination will be a pattern that we call a *timing fork*, which is similar to the one illustrated in Figure 1.

*Definition 11 (timing fork):* A timing fork in a computation $c$ is a triple $F = \langle \theta_0, \theta_1, \theta_2 \rangle$ of nodes in $c$, where there are signal delay chains $\theta_0 \rightsquigarrow_c \theta_1$ and $\theta_0 \rightsquigarrow_c \theta_2$. For such a timing fork, we denote $\mathbf{base}(F) = \theta_0$, $\mathbf{head}(F) = \theta_1$, and $\mathbf{tail}(F) = \theta_2$.

We define the *weight* of a timing fork, $\mathbf{wt}(F)$, to be

$$\mathbf{wt}(F) \overset{\text{def}}{=} \mathsf{lb}(\theta_0 \rightsquigarrow_c \theta_1) - \mathsf{ub}(\theta_0 \rightsquigarrow_c \theta_2)$$

Given a timing fork $F$ in computation $c$, it is clear that

$$\mathbf{time}_c(\mathbf{head}(F)) \geq \mathbf{time}_c(\mathbf{tail}(F)) + \mathbf{wt}(F)$$

simply from the definition of a computation and the time bounds; in other words, $\mathbf{tail}(F) \xrightarrow{\mathbf{wt}(F)} \mathbf{head}(F)$. Hence, a timing fork is a mechanism that can order two nodes in time by the weight of the fork.

A *zigzag pattern* consists of a collection of timing forks satisfying certain constraints. In a zigzag pattern, there is an ordering of forks so that the head of one fork and the tail of the next fork are nodes from the same process, with the head appearing no later than the tail. Figure 3 shows a zigzag pattern consisting of two timing forks.

*Definition 12 (zigzag):* A zigzag pattern from node $\theta$ to $\theta'$ in a computation $c$ is a sequence $Z = (F_1, \ldots, F_n)$ of forks in $c$ such that: (i) $\theta = \mathbf{tail}(F_1)$; (ii) $\theta' = \mathbf{head}(F_n)$; (iii) $\mathbf{head}(F_i) \leq^c_\ell \mathbf{tail}(F_{i+1})$ for all $1 \leq i < n$.

### A. Zigzags and Timing Information

The weight of a zigzag pattern $Z = (F_1, \ldots, F_n)$ is defined to be

$$\mathbf{wt}(Z) = \sum_{i=1}^{n} \mathbf{wt}(F_i)$$

Each timing fork in the zigzag pattern contributes the bound $\mathbf{tail}(F_i) \xrightarrow{\mathbf{wt}(F_i)} \mathbf{head}(F_i)$. Furthermore, because the forks are in a zigzag configuration, we know that for $1 \leq i < n$, $\mathbf{head}(F_i) \leq^c_\ell \mathbf{tail}(F_{i+1})$. Since $\mathbf{wt}(Z) = \sum_i \mathbf{wt}(F_i)$, we obtain $\mathbf{tail}(F_1) \xrightarrow{\mathbf{wt}(Z)} \mathbf{head}(F_n)$. Thus, the existence of such a zigzag pattern provides a time bound relating $\mathbf{tail}(F_1)$ with $\mathbf{head}(F_n)$. While a timing fork is the simplest form of a zigzag pattern, the novelty in the notion of zigzag patterns is that nodes may be ordered in time without the existence of a single timing fork that relates them.

A natural question is the following: do zigzag patterns capture all of the information about the time bound between two nodes? In the rest of this section, we show that zigzags are not just sufficient to guarantee time bounds (shown above), but also necessary. In other words, the only way to guarantee time bounds among nodes is via a zigzag pattern. Thus, in a precise sense, zigzags capture all relevant information about time bounds between events.

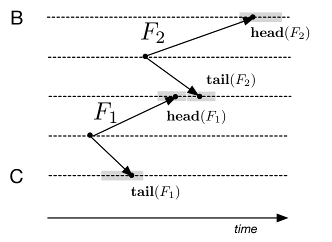

Fig. 3. A zigzag pattern consisting of two forks $F_1$ and $F_2$ that is sufficient to provide a lower bound on the time by which the arrival of the $F_2$ signal to $B$ occurs after the arrival of the $F_1$ signal to $C$.

### B. The Bounds Graph

Consider the timing fork in Fig. 1, where the gray regions correspond to the range of times when a signal from $A$'s node $\nu$ could be delivered to $B$ or $C$. Since processes in our model cannot directly determine the current time, different timings of signal arrival cannot be distinguished by a process. However, changes in relative signal arrival order can be observed locally by a process. To capture this, we introduce the notion of locally equivalent computations—computations that cannot be distinguished using local state information.

*Definition 13 (locally equivalent computations):* Computations $c$ and $d$ are said to be locally equivalent iff they have the same set of nodes and the happens-before relations $\prec_c$ and $\prec_d$ are the same.

Two locally equivalent computations only differ in their timing behavior. The timing behavior that they exhibit is constrained by the process and wire delay bounds in the TSP model. To capture this requirement, we introduce the notion of a *bounds graph* for a computation. We begin by defining the next non-absorbing node of a given node as follows:

*Definition 14 (next non-absorbing node):* Node $\nu'$ is said to be the next non-absorbing node of a node $\nu$ if $\nu'$ is the earliest node satisfying: (i) $\nu <^c_\ell \nu'$; and (ii) $\nu'$ is obtained from its predecessor by $\mathtt{proc}(\nu')$ following rules (**act**), (**reset**), or (**max-act**).

Armed with this definition, we proceed to define the bounds graph of a computation as follows:

*Definition 15 (bounds graph):* Let $c$ be a valid computation of a TSP system. The bounds graph of $c$ is the weighted graph $(V_c, E_c, \Delta_c)$, where $V_c$ is the set of nodes that appear in $c$, $E_c$ is a set of edges, and $\Delta_c$ maps each edge to an integer weight. The edges and weights are defined as follows: (B:i) Given node $\nu \in V_c$ and its next local node $\nu^+$, we add edge $(\nu, \nu^+)$ with weight 1; (B:ii) if $\nu \hookrightarrow_c \nu'$ due to a signal being delivered over wire $\mathsf{w}$, we add three edges. Let $\nu^+$ be the next local node of $\nu$ in $c$. We add edge $(\nu^+, \nu')$ with weight $\mathsf{lb}(\mathsf{w})$ and edge $(\nu', \nu^+)$ with weight $-\mathsf{ub}(\mathsf{w})$ to account for delay bounds on wires. We also add edge $(\nu', \nu'')$ with weight 1 where $\nu''$ is the earliest $\mathtt{proc}(\nu)$-node such that $\nu^+ \leq_\ell^c \nu''$ and in which the value of $o_\mathsf{w}$ at $\nu''$ differs from its value at $\nu^+$. We call this a positive acknowledgment edge.[5] (iii) Let $\nu$ be a non-quiescient $p$-node with $\mathsf{k}_p = 0$. There are two scenarios for the next non-absorbing node $\nu'$ following $\nu$: (B:iii-a) $\nu'$ is obtained by $p$ taking the pending action enabled in $\nu$, or (B:iii-b) $\nu'$ is obtained by $p$ taking action (**reset**) due to the arrival of a new input signal. In (B:iii-a), we add the edge $(\nu, \nu')$ with weight $\mathsf{lb}(p)$, and the edge $(\nu', \nu)$ with weight $-\mathsf{ub}(p)$. In (B:iii-b), we add the edge $(\nu', \nu)$ with weight $-(\mathsf{ub}(p) - 1)$.[6] (B:iv) If (a) a pending signal delivery on $\mathsf{w}$ from process $p$ to $q$ initiated at $\nu'_p$ is cancelled by process $p$ at time $t$, (b) $\nu_p$/$\nu_q$ is the $p$/$q$-node in configuration $c_t$, and (c) $\nu_q^+$ is the next local node for $q$ after time $t$ in $c$, then we add "negative acknowledgment" edges: $(\nu_p, \nu_q^+)$ of weight 1, $(\nu_p, \nu'_p)$ of weight $-(\mathsf{ub}(\mathsf{w}) - 1)$, $(\nu_q, \nu_p)$ of weight 0, and $(\nu_p, \nu_q)$ of weight 0 if $\mathbf{time}_c(\nu_q) = t$.

These edges each capture constraints introduced by the definition of a valid computation in Section II-D.

*Lemma 2:* Let $c$ be a valid computation with bounds graph $(V_c, E_c, \Delta_c)$. Then $\mathbf{time}_c(\nu') \geq \mathbf{time}_c(\nu) + \Delta_c(\nu, \nu')$ whenever $(\nu, \nu') \in E_c$.

**Proof:** We show this by case analysis on the edges introduced in Definition 15. Time bound requirements introduced by edges due to clause (B:i) in Definition 15 are always satisfied, since distinct nodes for the same process that are ordered by "$\prec_c$" have times that are at least one apart. Those introduced by clause (B:ii) are satisfied because of the time bounds on wires. Those introduced by (B:iii-a) edges are satisfied because of the time bounds on process steps. Finally, (B:iii-b) edges are satisfied because a (**reset**) action can occur only before the process timer reaches its upper bound. □

The key idea behind the bounds graph is to capture the constraints on possible timings of a valid computation. The next lemma shows that this is the case. As a reminder, we have defined a computation as simply a sequence of configurations (Section II-D); only *valid* computations satisfy the process firing rules and timing constraints.

*Lemma 3 (equivalence):* Let $c$ be a valid computation of a TSP system $\mathcal{S}$. Let $d$ be any computation that has the same set of nodes, and whose timing is consistent with $c$'s bounds graph. Then: (i) $d$ is a valid computation of $\mathcal{S}$; and (ii) $c$ and $d$ are locally equivalent.

**Proof:** For any $p$ node $\nu$ in $c$ (and hence also in $d$), the next local node $\nu^+$ in $c$ must be the same as the next local node $\nu^+$ in $d$ because of the bounds graph edges introduced in case (B:i) of Definition 15. We call this property *local consistency*. In particular, this implies that both $c$ and $d$ have the same earliest $p$-node for any process $p$.

Part (i)—$d$ is a valid computation: Assume toward a contradiction that $d$ is not a valid computation of $\mathcal{S}$. If we examine the sequence of configurations $d_0$, $d_1$, …, then there is some earliest $d_j$ where the computation transitions from valid to being invalid. We must have $j > 0$ since every process has the same node in $c$ and $d$ in the initial configuration becase they have the same earliest $p$-node for any process $p$. For $d_{j-1}$ to be valid, but $d_j$ to be invalid, there must be some process $p$ that has a local state in $d_j$ that cannot be obtained from $d_{j-1}$ applying the rules in Section II-D. Let $\nu$ be the $p$-node in $d_{j-1}$, and $\nu'$ be the $p$-node in $d_j$ that cannot be obtained by applying the rules in Section II-D. We examine each possibility for both process and wire rule violations in $d_j$ given that $d_0, \dots, d_{j-1}$ is a valid computation prefix.

Suppose $\nu \neq \nu'$. Then $\nu'$ is the next local node for $\mathtt{proc}(\nu)$ in $c$ by local consistency. We first show that any changes in the controllable state in $\mathtt{proc}(\nu)$ must be valid in $d_j$. Any such change must conform to the process rules (because $c$ is a valid computation), and be caused by either (**act**) or (**max-act**) process rules. Since the bounds graph constrains the delay between nodes in $d$ (cases (B:i) and (B:iii-a)), any controllable state change in $\nu'$ remains valid in $d_j$. The other option is that $\nu'$ has an input change that was valid in $c$ but not in $d$. For an input to change, it means that a signal was delivered on a wire in $c$ to $\mathtt{proc}(\nu')$, from another process $q$, or a signal delivery was cancelled causing the input to be "?." The $q$-node in $d$ must appear before time $j$, because of the bounds graph constraint (case (B:ii) and (B:iv)) on signals delivered in $c$, and hence, it appears in $d$ prior to time $j$ as part of the valid computation prefix. The bounds graph constraint on wires implies that the same signal can be delivered or is cancelled over the same wire in $d_j$—showing that $\nu'$ is a valid computation node in $d_j$. Now suppose $\nu' = \nu$. If $\nu' = \nu$ is not valid, then it is because some forced state change did not occur in $d_j$. The only forced changes are (a) (**max-act**) for $\mathtt{proc}(\nu)$, which is again ruled out because of the bounds graph (case (B:iii-a)); (b) (**max-act**) for wires, which is also ruled out by the bounds graph (case (B:ii, B:iv)); (c) An action in another process canceled a signal delivery that should have occurred—which is ruled out as well by the acknowledgment edges (case (B:ii) and (B:iv)). Hence, we arrive at a contradiction, and conclude that $d$ is a valid computation of $\mathcal{S}$.

Part (ii)—local equivalence: Given local consistency, we only need to show that $\nu \hookrightarrow_c \nu'$ iff $\nu \hookrightarrow_d \nu'$. By the argument

[5]This ensures that the signal is delivered over $\mathsf{w}$. If the value of $o_\mathsf{w}$ changes before the signal is delivered, then the signal delivery would be cancelled—case (B:iv). The introduction of this edge in the bounds graph captures the fact that the change in $o_\mathsf{w}$ occurs after the signal is delivered over $\mathsf{w}$. The node $\nu''$ may not exist in $c$, in which case this edge is not added.

[6]This edge captures the constraint that a reset step can only occur before $\mathsf{k}_p$ reaches the upper bound on the local action delay in $p$.

we completed for Part (i), the same set of signals are delivered or cancelled in $d$ as they are in $c$, and they have the same source and target nodes in both $c$ and $d$. $\square$

Lemma 3 shows that the bounds graph captures all possible timing assignments for locally equivalent computations.

## C. Zigzag Necessity

*Definition 16 (timed coordination):* We say $\mathbf{Late}\langle\theta \xrightarrow{x} \theta'\rangle$ holds in a given a TSP network $\mathcal{S}$ iff every computation $c$ of $\mathcal{S}$ where both $\theta$ and $\theta'$ appear satisfies $\theta \xrightarrow{x} \theta'$.

*Theorem 1 (Zigzag necessity):* Let $\mathcal{S}$ be a TSP network where $\mathbf{Late}\langle\theta \xrightarrow{x} \theta'\rangle$ holds, and let $c$ be a valid computation of $\mathcal{S}$ where nodes $\theta$ and $\theta'$ appear. Then either there must be a zigzag pattern with weight at least $x$ in $\mathcal{S}$, or a process that can enter the erroneous state is involved in ordering $\theta$ and $\theta'$.
**Proof:** Consider any valid computation $c$ where both $\theta$ and $\theta'$ appear. We know that any computation $d$ that has the same set of nodes as in $c$ and respects $c$'s bounds graph is a valid computation by Lemma 3. We select $d$ such that $\theta'$ occurs as early as possible, and given the earliest possible time for $\theta'$, $\theta$ occurs as late as possible. The reason $\theta = \theta_0$ cannot be moved to a later time must be because it is constrained by the bounds graph—in particular, by an edge to another node $\theta_1$. If the edge has positive weight, then $\theta_1$ occurs after $\theta_0$; if the edge has negative weight, then $\theta_1$ occurs before $\theta_0$. We then consider why $\theta_1$ cannot be moved forward in time, and repeat this argument until we eventually reach node $\theta_n = \theta'$. Note that if we don't reach $\theta'$ that is fixed in time, we can translate $\theta = \theta_0, \theta_1, \ldots$, etc. forward in time until we violate $\theta \xrightarrow{x} \theta'$ in $d$—a contradiction.

Examining the sequence of bounds graph edges constraining $\theta$, either (a) none of them are of type (B:iii-b) or an acknowledgment edge in (B:ii), and no node along the path is the result of a (**reset**) step. In this case, these edges form a zigzag pattern; or (b) at least one of the edges is of type (B:iii-b) or an acknowledgment edge in (B:ii, B:iv). In this case, we can make a process/wire enter the $\perp$/? state, as these edges were introduced in the bounds graph construction but do not arise due to any constraints on a valid computation. $\square$

## V. Discussion

We have shown that timing forks are the essential construct needed to reason about timing information among signal transitions in the timed asynchronous setting. A commonly used way to express timing requirements in circuit design is via a point-of-divergence (pod) constraint. A pod constraint has a root signal transition and two targets, where each path from the root to one of the targets is always supposed to be faster than every path to the other target. In our framework, a point-of-divergence constraint is captured directly by a timing fork (a **pod-**timing fork). In this particular context, it is convenient to refer to the two signal delay chains of the timing fork as two *tines*: a *slow tine*, and a *fast tine*, where the slow tine must always be slower than the fast tine.

A special case of the timed asynchronous setting is *synchronous logic*, where a single global timing signal is used to orchestrate the computation. Standard edge-triggered synchronous logic has two constraints for correctness: (i) the *setup time* constraint, which states that the delay from the outputs of flip-flops through the combinational logic to the inputs of flip-flops must be smaller than the clock period; and (ii) *hold time* constraints, where the minimum delay through the combinational logic is larger than the relative arrival times of the clock pins. Figure 4 shows that both these constraints are in fact pod-timing forks. The roots of both forks correspond to the zero-to-one transition of the clock. In traditional synchronous design, these are viewed as distinct types of timing constraints (hence the two different names). Our framework shows that they are in fact two instances of the more general notion of a timing fork.

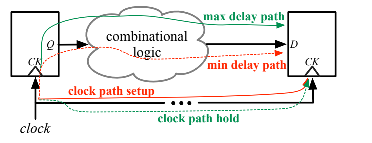

Fig. 4. The setup time constraint for clocked logic is a timing fork depicted by solid arrows where the clock path setup (slow tine) must be slower than the max delay path (fast tine) with a weight that corresponds to roughly the clock period. The hold time constraint is also a timing fork depicted by dashed arrows where the min delay path (slow tine) must be slower than the clock path hold (fast tine) with a weight that is roughly zero.

It is interesting to note that a violation of a pod-timing fork constraint can be viewed in two ways: we can either say that the slow tine is too fast, or that the fast tine is too slow. For the setup pod-timing fork, the former corresponds to the clock period being too small, and the latter corresponds to the worst-case logic delay being too large. For the hold pod-timing fork, the former corresponds to the minimum delay through the logic being too small, while the latter corresponds to too much relative clock skew between the two flip-flops.

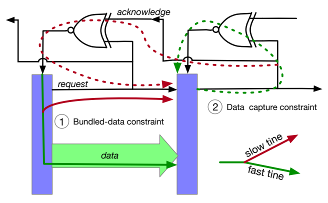

Fig. 5. The timing forks required for correct operation of a MOUSETRAP pipeline. The green arrow is the fast tine, the red arrow is the slow tine. Each fork shown corresponds to two timing forks: one for the zero-to-one transition and one for the one-to-zero transition.

We can illustrate pod-timing forks in high-speed asynchronous pipelines using the MOUSETRAP transition signalling asynchronous pipeline template. Figure 5 shows two adjacent MOUSETRAP FIFO stages. Initially, the latches are transparent. When a new input arrives, indicated by the request toggling, two signal transition paths are activated: (i) the path through the local XNOR gate that causes the capturing latch to close (the green dashed line in Figure 5), and the acknowledge wire on the red dashed line in Figure 5. The timing constraint required for correct operation is that the local XNOR path must close the capturing latch before the acknowledge path has a chance to send the next request. The corresponding pod-timing fork is shown in Figure 5. The second consraint is the data bundling constraint, which is another pod-timing fork in Figure 5. Other work has used pod-timing forks to specify timing constraints for many asynchronous circuit families including standard bundled-data micropipelines, relaxed QDI circuits, scalable delay-insensitive circuits, GasP circuits, and single-track full buffers [11].

There is a gap between Theorem 1 and the sufficiency of zigzag patterns in reasoning about signal transition ordering. The path constructed in Theorem 1 could include edges in the bounds graph that correspond to delay bounds in processes and edges, but can also include acknowledgment edges from case (B:ii)/(B:iv) or reset edges from case (B:iii-b) that show that the computation can enter a $\perp$ state. Recall that such computations involve glitches/erroneous behavior. When such edges are present, we call the resulting structure a *generalized zigzag pattern*. As the behavior of such erroneous scenarios is challenging to reason about even with a detailed description of the underlying implementation, a common practice in circuit design is to rely on timing constraints between signals that hazard-free. In this scenario, a zigzag pattern is a necessary condition by Theorem 1.

A second use of timing information is when one tine of a pod-timing fork is hazard-free, but the other tine can have switching hazards. This occurs, for example, in bundled-data protocols. In this case, there are mutiple forks corresponding to all the possible nodes that appear in a computation corresponding to all the different hazard scenarios. The timing constraints to be satisfied are a function of the set of nodes that appear in the computation. The slowest path through the hazardous logic in fact does not have switching hazards, as it coresponds to the *last* input signal change for each gate along the path; hence, even in this scenario, Theorem 1 shows that the only way to guarantee the constraint is via a zigzag pattern.

While a circuit might rely on a generalized zigzag pattern for ordering transitions, such a circuit is subject to unstable behavior—and is likely to be erroneous for other reasons.

*Related work:* In the untimed setting, the closest work to the TSP model is the asynchronous signalling processes (SP) model [6]. In the SP model, delays are finite but unbounded, and signal transitions can only be ordered in time via explicit signalling chains. The TSP model can be viewed as an extension of the SP model where timing information is used.

In the timed setting, there are a number of efforts that model asynchronous circuits using upper and lower bounds on the delays of gates and wires in the literature. Broadly speaking, these efforts provide methods to compute the reachable state-space and next-state functions of an asynchronous circuit using timing information to potentially prune the set of reachable logical states. This timed state space is used for either circuit synthesis [5], or for verification of properties of asynchronous circuits [12]. The underlying models used include timed event structures, timed traces, or timed Petri nets [5], [12], [13].

Relative timing constraints are assertions that say that two signal transitions are ordered [14]. Such constraints have been shown to be useful in developing optimized versions of asynchronous circuits, and have also been incorporated into recent versions of circuit synthesis tools like Petrify [15].

The TSP model also uses upper and lower bounds to model the realized timing behavior of an asynchronous circuit. Our key contribution is showing that zigzag patterns are both sufficient as well as necessary conditions for ordering signal transitions in the timed asynchronous setting via theoretical analysis, which provides an insight into how relative timing constraints must be realized in practice.

Our work is closely related to the distributed systems setting where communicating processes interact via message-passing channels, and where there are known time bounds for message propagation on channels [7]. Dan et. al's work [7] introduces zigzag patterns in the message-passing context under a flooding full-information protocol assumption. They show that message zigzags are necessary and sufficient for ordering process actions. In the TSP model, processes interact via wires and are not constrained by the flooding full-information protocol. Nonetheless, our work shows that signal delay chains play the same role as message chains do in the distributed systems setting, and signal zigzags are necessary and sufficient to order signal transitions. This provides another illustration of the deep connection between distributed systems and asynchronous circuits in the timed setting.

## VI. Summary

We introduced the timed signalling processes (TSP) model that captures the behavior of a large class of digital systems including clocked circuits as well as asynchronous circuits that may exploit delay information for their operation. We defined two communication structures: the timing fork, and a zigzag pattern (a collection of timing forks in a specific configuration). We illustrated how these structures capture commonly used timing requirements in different digital circuit families. Most importantly, we proved that zigzag patterns capture all the usable timing information among signal transitions in the TSP model.

## References

[1] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proc. ARVLSI*, pp. 263–278, MIT Press, 1990.

[2] I. Sutherland, "Micropipelines," *CACM*, vol. 32, no. 6, pp. 720–738, 1989.

[3] M. Singh and S. M. Nowick, "Mousetrap: Ultra-high-speed transition-signaling asynchronous pipelines," in *Proc. ICCD*, pp. 9–17, 2001.

[4] I. Sutherland and S. Fairbanks, "Gasp: A minimal fifo control," in *Proc. ASYNC*, pp. 46–53, 2001.

[5] C. J. Myers and T. Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE TVLSI*, 1993.

[6] R. Manohar and Y. Moses, "Asynchronous signalling processes," in *Proc. ASYNC*, 2019.

[7] A. Dan, R. Manohar, and Y. Moses, "On using time without clocks via zigzag causality," in *ACM PODC*, pp. 241–250, ACM, 2017.

[8] Y. Moses and B. Bloom, "Knowledge, timed precedence and clocks (preliminary report)," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pp. 294–303, 1994.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] R. Manohar and Y. Moses, "Analyzing isochronic forks with potential causality," in *Proc. ASYNC*, pp. 69–76, IEEE, 2015.

[11] R. Dashkin and R. Manohar, "General approach to asynchronous circuits simulation using synchronous fpgas," *IEEE TCAD*, vol. 41, no. 10, pp. 3452–3465, 2022.

[12] T. G. Rokicki and C. J. Myers, "Automatic verification of timed circuits," in *Proc. CAV*, pp. 468–480, Springer, 1994.

[13] P. Merlin and D. Farber, "Recoverability of communication protocols-implications of a theoretical study," *IEEE transactions on Communications*, vol. 24, no. 9, pp. 1036–1043, 1976.

[14] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative timing," *IEEE TVLSI*, vol. 11, no. 1, pp. 129–140, 2003.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on information and Systems*, vol. 80, no. 3, pp. 315–325, 1997.