Fault-tolerant Algorithms for Tick-Generation in Asynchronous Logic: Robust Pulse Generation [Extended Abstract]*

Danny Dolev[1], Matthias Függer[2], Christoph Lenzen[1], and Ulrich Schmid[2]

[1] Hebrew University of Jerusalem, Jerusalem, Israel
{dolev,clenzen}@cs.huji.ac.il
[2] Vienna University of Technology, Vienna, Austria
{fuegger,s}@ecs.tuwien.ac.at

Abstract. The advances of deep submicron VLSI technology pose new challenges in designing robust systems, which can in principle be addressed by approaches established in fault-tolerant distributed systems research. This paper is the first step in an attempt to develop a very robust high-precision clocking system for hardware designs like systems-on-chip for critical applications. It is devoted to the design and the correctness proof of a novel Byzantine fault-tolerant self-stabilizing pulse synchronization protocol, which facilitates a direct implementation in standard asynchronous digital logic. Despite the severe implementation constraints, it offers optimal resilience and smaller complexity than all existing pulse synchronization protocols.

Keywords: clock synchronization, Byzantine faults, self-stabilization

1 Introduction & Related Work

With today's deep submicron technology running at GHz clock speeds [19], disseminating the high-speed clock throughout a *very large scale integrated* (VLSI) circuit, with negligible skew, is difficult and costly [2,3,11,23,27]. Systems-on-chip are hence increasingly designed *globally asynchronous locally synchronous* (GALS) [4], where different parts of the chip use different local clock signals. Two main types of clocking schemes for GALS systems exist, namely, (i) those where the local clock signals are unrelated, and (ii) multi-synchronous ones that provide a certain degree of synchrony between local clock signals [28,31].

GALS systems clocked by type (i) permanently bear the risk of *metastable upsets* when conveying information from one clock domain to another. To explain the issue, consider a physical implementation of a bistable storage element, like a register cell, which can be accessed by read and write operations concurrently.

* The full paper is available at the arxiv [9]. This work has been supported by the Swiss National Science Foundation (SNSF), by the Austrian Science Foundation (FWF) project FATAL (P21694), and by by the Israeli Science Foundation (ISF) Grant number 1685/07. Danny Dolev is Incumbent of the Berthold Badler Chair.

It can be shown that two operations occurring very closely to each other can cause the storage cell to attain neither of its two stable states for an unbounded time [22]. Although the probability of a single upset is very small, one has to take into account that every bit of transmitted information across clock domains is a candidate for an upset. Elaborate synchronizers [8,20,26] are the only means for achieving an acceptably low probability for metastable upsets here.

This problem can be circumvented in clocking schemes of type (ii): Common synchrony properties offered by multi-synchronous clocking systems are bounded *precision*, i.e., bounded maximum offset in the number of clock transitions of any two local clock signals, and bounded *accuracy*, i.e., bounded difference of the local clock rate and the rate of progress of real time. Type (ii) clocking schemes are particularly beneficial from a designers point of view, since they combine the convenient local synchrony of a GALS system with a global time base across the whole chip. It has been shown in [25] that these properties indeed facilitate metastability-free high-speed communication across clock domains.

The decreasing structure sizes of deep submicron technology also resulted in an increased likelihood of chip components failing during operation: Reduced voltage swing and smaller critical charges make circuits more susceptible to ionized particle hits, crosstalk, and electromagnetic interference [5,17]. *Fault-tolerance* hence becomes an increasingly pressing issue in chip design. Unfortunately, faulty components may behave non-benign in many ways. They may perform signal transitions at arbitrary times and even convey inconsistent information to their successor components if their outgoing communication channels are affected by a failure. This forces to model faulty components as unrestricted, i.e., Byzantine, if a high fault coverage is to be guaranteed.

The DARTS fault-tolerant clock generation approach [14,16] developed by some of the authors of this paper is a Byzantine fault-tolerant multi-synchronous clocking scheme. DARTS comprises a set of modules, each of which generates a local clock signal for a single clock domain. The DARTS modules (nodes) are synchronized to each other to within a few clock cycles. This is achieved by exchanging binary clock signals only, via single wires. The basic idea behind DARTS is to employ a simple fault-tolerant distributed algorithm [32]—based on Srikanth & Toueg's consistent broadcasting primitive [29]—implemented in asynchronous digital logic. An important property of the DARTS clocking scheme is that it guarantees that no metastable upsets occur during fault-free executions. For executions with faults, metastable upsets cannot be ruled out: Since Byzantine faulty components are allowed to issue unrelated read and write accesses by definition, the same arguments as for clocking schemes of type (i) apply. However, in [12], it was shown that the probability of a Byzantine component leading to a metastable upset of DARTS can be made arbitrarily small.

Although both theoretical analysis and experimental evaluation revealed many attractive additional features of DARTS, like guaranteed startup, automatic adaption to current operating conditions, etc., there is room for improvement. The most obvious drawback of DARTS is its inability to support late joining and restarting of nodes, and, more generally, its lack of self-stabilization prop-

erties. If, for some reasons, more than a third of the DARTS nodes ever become faulty, the system cannot be guaranteed to resume normal operation even if all failures cease. Even worse, simple transient faults such as radiation- or crosstalk-induced additional (or omitted) clock ticks accumulate over time to arbitrarily large skews in an otherwise benign execution.

Byzantine-tolerant self-stabilization, on the other hand, is the major strength of a number of protocols [1,6,10,18,21] primarily devised for distributed systems. Of particular interest in the above context is the work on self-stabilizing *pulse synchronization*, where the purpose is to generate well-separated anonymous pulses that are synchronized at all correct nodes. This facilitates self-stabilizing clock synchronization, as agreement on a time window permits to simulate a synchronous protocol in a bounded-delay system. Beyond optimal (i.e., $\lceil n/3 \rceil - 1$, c.f. [24]) resilience, an attractive feature of these protocols is a small stabilization time [1,6,18,21], which is crucial for applications with stringent availability requirements. In particular, [1] synchronizes clocks in expected constant time in a synchronous system. Given any pulse synchronization protocol stabilizing in a bounded-delay system in expected time T, this implies an expected $(T + \mathcal{O}(1))$-stabilizing clock synchronization protocol.

Note that existing synchronization algorithms, in particular those that do not rely on pulse synchronization, have deficiencies rendering them unsuitable in our context. For example, they have exponential convergence time [10], require the relative drift of the nodes' local clocks to be very small [7,21], provide low synchronization precision [21] or make use of linear-sized messages [6]. Furthermore, standard distributed systems' models do not account for metastability.

In this paper, we describe and prove correct the novel FATAL pulse synchronization protocol, which facilitates a direct implementation in standard asynchronous digital logic. It self-stabilizes within $\mathcal{O}(n)$ time with probability $1 - 2^{n-f}$, in the presence of up to $f = \lceil n/3 \rceil - 1$ Byzantine faulty nodes, and is metastability-free by construction after stabilization in failure-free runs. While executing the protocol, non-faulty nodes broadcast a constant number of bits in constant time. In terms of distributed message complexity, this implies that stabilization is achieved after broadcasting $\mathcal{O}(n)$ messages of size $\mathcal{O}(1)$, improving by factor $\Omega(n)$ on the number of bits transmitted by previous algorithms.[3] The protocol can sustain large relative clock drifts of more than 10%, which is crucial if the local clock sources are simple ring oscillators (uncompensated ring oscillators suffer from clock drifts of up to 9% [30]). If the number of faults is not overwhelming, i.e., a majority of at least $n - f$ nodes continues to execute the protocol in an orderly fashion, recovering nodes and late joiners (re)synchronize in constant time. All this is achieved against a powerful adversary that, at time t, knows the whole history of the system up to time $t + \varepsilon$ (where $\varepsilon > 0$ is infinitesimally small) and does not need to choose the set of faulty nodes in advance.

[3] We remark that [21] achieves the same complexity, but considers a much simpler model. In particular, *all* communication is restricted to broadcasts, i.e., all nodes observe the same behaviour of a given other node, even if it is faulty.

2 Model

Our formal framework will be tied to the peculiarities of hardware designs, which consist of modules that *continuously*[4] compute their output signals based on their input signals. Following [13,15], we define (the trace of) a *signal* to be a timed event trace over a finite alphabet $\mathbb{S}$ of possible signal states: Formally, signal $\sigma \subseteq \mathbb{S} \times \mathbb{R}_0^+$. All times and time intervals refer to a global *reference time* taken from $\mathbb{R}_0^+$, that is, signals describe the system's behavior from time 0 on. The elements of σ are called *events*, and for each event (s,t) we call s the *state of event* (s,t) and t the *time of event* (s,t). In general, a signal σ is required to fulfill the following conditions: (i) for each time interval $[t^-,t^+] \subseteq \mathbb{R}_0^+$ of finite length, the number of events in σ with times within $[t^-,t^+]$ is finite, (ii) from $(s,t) \in \sigma$ and $(s',t) \in \sigma$ follows that $s = s'$, and (iii) there exists an event at time 0 in σ.

Note that our definition allows for events (s,t) and $(s,t') \in \sigma$, where $t < t'$, without having an event $(s'',t'') \in \sigma$ with $s'' \neq s$ and $t < t'' < t'$. In this case, we call event (s,t') *idempotent*. Two signals σ and σ' are *equivalent*, iff they differ in idempotent events only. We identify all signals of an equivalence class, as they describe the same physical signal. Each equivalence class $[\sigma]$ of signals contains a unique signal σ_0 having no idempotent events. We say that *signal σ switches to s* at time t iff event $(s,t) \in \sigma_0$. The *state of signal* σ at time $t \in \mathbb{R}_0^+$, denoted by $\sigma(t)$, is given by the state of the event with the maximum time not greater than t.[5] Because of (i), (ii) and (iii), $\sigma(t)$ is well defined for each time $t \in \mathbb{R}_0^+$. Note that σ's state function in fact depends on $[\sigma]$ only, i.e., we may add or remove idempotent events at will without changing the state function.

Distributed System. A distributed system is a finite set of n nodes $V = \{1,\dots,n\}$. Each node i comprises a number of *input ports*, namely $S_{i,j}$ for each node j, an *output port* S_i, and a set of *local ports*, introduced later on. An *execution* of the distributed system assigns to each port of each node a signal. For convenience of notation, for any port p, we refer to the signal assigned to port p simply by signal p. We say that *node i is in state s* at time t iff $S_i(t) = s$ and that *node i switches to state s* at time t iff signal S_i switches to s at time t.

To enable nodes to communicate (that is, exchange their state), we assume the existence of *channels* between them: for each pair of nodes i,j, output port S_i is connected to input port $S_{j,i}$ by a FIFO channel from i to j with *maximum delay* $d > 0$.[6] Note that this includes a channel from i to i itself. The *channel* from node i to j is said to be *correct* during $[t^-,t^+]$ iff there exists a function $\tau_{i,j} : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, called the channel's *delay function*, such that: (i) $\tau_{i,j}$ is continuous and strictly increasing, (ii) $\forall t \in [t^-,t^+] : 0 \leq \tau_{i,j}(t) - t < d$, and (iii) for each $t \in [t^-,t^+]$, $(s,t) \in S_{j,i} \Leftrightarrow (s,\tau_{i,j}^{-1}(t)) \in S_i$, where τ^{-1} is the inverse of $\tau|_{[t^-,t^+]}$, i.e., τ restricted to $[t^-,t^+]$. Node i *observes node j in state s* at time t if $S_{i,j}(t) = s$.

[4] In sharp contrast to classic distributed computing models, there is no computationally complex discrete zero-time state-transition here.

[5] Whenever referring to σ, we will talk of the signal, not the state function.

[6] W.r.t. $\mathcal{O}$-notation, we normalize $d \in \mathcal{O}(1)$, as all time bounds depend linearly on d.

Clocks and Timeouts. Nodes are never aware of the current reference time and we also do not require it to resemble Newtonian "real" time. Rather we allow for physical clocks that run arbitrarily fast or slow, as long as their speeds are close to each other in comparison. One may hence think of the reference time as progressing at the speed of the currently slowest correct clock. In this framework, nodes essentially make use of bounded clocks with bounded drift.

Formally, clock rates are within $[1, \vartheta]$ (with respect to reference time), where $\vartheta > 1$ is constant and $\vartheta - 1$ is the *(maximum) clock drift*. A *clock* C is a continuous, strictly increasing function $C : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ mapping reference time to local time. Clock C is said to be *correct* during $[t^-, t^+] \subseteq \mathbb{R}_0^+$ iff we have for any $t, t' \in [t^-, t^+]$, $t < t'$, that $t' - t \leq C(t') - C(t) \leq \vartheta(t' - t)$. Each node comprises a set of clocks assigned to it, which allow the node to estimate the progress of reference time.

Instead of directly accessing their clocks, nodes have so-called *timeout ports* of watchdog timers. A *timeout* is a triple (T, s, C), where $T \in \mathbb{R}^+$, $s \in \mathbb{S}$, and C is a clock, say of node i. Each timeout (T, s, C) has a corresponding timeout port $\mathrm{Time}_{T,s,C}$, being part of node i's local ports. Signal (T, s, C) is Boolean, that is, its possible states are from the set $\{0, 1\}$. We say that timeout (T, s, C) is *correct* during $[t^-, t^+] \subseteq \mathbb{R}_0^+$ iff clock C is correct during $[t^-, t^+]$ and the following holds:

1. For each time $t_s \in [t^-, t^+]$ when node i switches to state s, there is a time $t \in [t_s, \tau_{i,i}(t_s)]$ such that (T, s, C) is *reset*, i.e., $(0, t) \in \mathrm{Time}_{T,s,C}$. This is a one-to-one correspondence, i.e., (T, s, C) is not reset at any other times.
2. For a time $t \in [t^-, t^+]$, denote by t_0 the supremum of all times from $[t^-, t]$ when (T, s, C) is reset. Then it holds that $(1, t) \in \mathrm{Time}_{T,s,C}$ iff $c(t) - c(t_0) = T$. Again, this is a one-to-one correspondence.

We say that timeout (T, s, C) *expires* at time t iff $\mathrm{Time}_{T,s,C}$ switches to 1 at time t, and it *is expired* at time t iff $\mathrm{Time}_{T,s,C}(t) = 1$. We will omit the clock C from the notation and simply write (T, s) for both the timeout and its signal.

A *randomized timeout* is a triple $(\mathcal{D}, s, C)$, where $\mathcal{D}$ is a bounded random distribution on $\mathbb{R}_0^+$, $s \in \mathbb{S}$ is a state, and C is a clock. Its corresponding timeout port $\mathrm{Time}_{\mathcal{D},s,C}$ behaves very similar to the one of an ordinary timeout, except that whenever it is reset, the local time that passes until it expires next—provided that it is not reset again before that happens—follows the distribution $\mathcal{D}$. For the purpose of this abstract, we give a simplified formal definition here. A randomized timeout $(\mathcal{D}, s, C)$ is correct during $[t^-, t^+] \subseteq \mathbb{R}_0^+$, if C is correct during $[t^-, t^+]$ and the following holds:

1. For each time $t_s \in [t^-, t^+]$ when node i switches to state s, there is a time $t \in [t_s, \tau_{i,i}(t_s)]$ such that $(\mathcal{D}, s, C)$ is *reset*, i.e., $(0, t) \in \mathrm{Time}_{\mathcal{D},s,C}$. This is a one-to-one correspondence, i.e., $(\mathcal{D}, s, C)$ is not reset at any other times.
2. For a time $t \in [t^-, t^+]$, denote by t_0 the supremum of all times from $[t^-, t]$ when $(\mathcal{D}, s, C)$ is reset. Let $\mu : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ denote the density of $\mathcal{D}$. Then

$$P[\mathrm{Time}_{\mathcal{D},s,C} \text{ switches to 1 during } [t_0, t]] = \int_{t_0}^{t} \mu(C(t) - C(t_0))\, d\tau.$$

We will apply the same notational conventions to randomized timeouts as we do for regular timeouts.

We remark that these definitions allow for different timeouts to be driven by the same clock, implying that an adversary may derive some information on the state of a randomized timeout before it expires from the node's behaviour, even if it cannot directly access the values of the clock driving the timeout. This is crucial for efficient implementability, as nodes require one clock only.

Memory Flags Another kind of node i's local ports are *memory flags*. For each state $s \in \mathbb{S}$ and each node $j \in V$, memory flag $\mathrm{Mem}_{i,j,s}$ is a local port of node i. It is used to memorize whether node i has observed node j in state s since the last reset of the flag. We say that node i *memorizes node j in state s* at time t if $\mathrm{Mem}_{i,j,s}(t) = 1$. Formally, we require that signal $\mathrm{Mem}_{i,j,s}$ switches to 1 at time t iff node i observes node j in state s at time t and $\mathrm{Mem}_{i,j,s}$ is not already in state 1. The times when $\mathrm{Mem}_{i,j,s}$ is *reset*, i.e., when signal $\mathrm{Mem}_{i,j,s}$ switches to 0, are specified by node i's state machine, which is introduced next.

State Machine It remains to specify how nodes switch states and when they reset memory flags. We do this by means of state machines that may attain states from the finite alphabet $\mathbb{S}$. Node i's state machine is specified by (i) the set $\mathbb{S}$, (ii) a function tr, called the *transition function*, from $\mathcal{T} \subseteq \mathbb{S}^2$ to the set of Boolean predicates on the alphabet consisting of expressions "$p = s$" (used for expressing guards), where p is a local or input port of i and s is a possible state of signal p, and (iii) a function re, called the *reset function*, from $\mathcal{T}$ to the power set of the node's memory flags.

Intuitively, the transition function specifies the conditions (guards) under which a node switches states, and the reset function determines which memory flags to reset upon the state change. Formally, let P be a predicate on node i's input and local ports. We define P *holds at time t* by structural induction: If P is an element of the above alphabet, i.e., $p = s$, then P *holds at time t* iff $p(t) = s$. Otherwise, if P is of the form $\neg P_1$, $P_1 \wedge P_2$, or $P_1 \vee P_2$, we define P *holds at time t* in the straightforward manner.

We say node i *follows its state machine during* $[t^-, t^+]$ iff the following holds: Assume node i observes itself in state $s \in \mathbb{S}$ at time $t \in [t^-, t^+]$, i.e., $S_{i,i}(t) = s$. Then, for each $(s, s') \in \mathcal{T}$, both:

1. Node i switches to state s' at time t iff $tr(s, s')$ holds at time t and i is not already in state s'.[7]
2. Node i resets memory flag m at some time within $[t, \tau_{i,i}(t)]$ iff $m \in re(s, s')$ and i switches to state s' at time t. This correspondence is one-to-one.

A node may also run several state machines in parallel. In this case, S_i simply is the product of the individual machine's output signals, and the different state machines interact by means of the delayed signal $S_{i,i}$ only.

A node is defined to be *non-faulty* during $[t^-, t^+]$ iff during $[t^-, t^+]$ all its timeouts and randomized timeouts are correct and it follows (all of) its state machine(s). In contrast, a faulty node may change states arbitrarily. While a

[7] If more than one guard $tr(s, s')$ can be true concurrently, break ties arbitrarily.

faulty node may be forced to send consistent states to all other nodes if its channels remain correct, there is no way to guarantee that this still holds if channels are faulty.[8]

Metastability In our discrete system model, the effect of metastability is captured by the lacking capability of state machines to instantaneously take on new states: Node i decides on state transitions based on the delayed status of port $S_{i,i}$ instead of its "true" current state S_i. This non-zero delay from S_i to $S_{i,i}$ bears the potential for metastability, as a successful state transition can only be guaranteed if the transition guard remains stable during this delay at least. Hence, we define that node $i \in V$ is *metastability-free during* $[t^-, t^+]$ *with respect to one of its state machines* M, iff for any time $t \in [t^-, t^+]$ when i switches to a state s of M, the infimum t' of times in $(t, t^+]$ when i switches to some state s' of M satisfies $t' > \tau_{i,i}(t)$.

3 The FATAL Pulse Synchronization Protocol

In this section, we present our self-stabilizing pulse generation algorithm, which will be stated in terms of state machines, as introduced in the previous section. Its goal is to generate synchronized, well-separated pulses that occur upon switching to a distinguished state *accept*. For a set of nodes $W \subseteq V$ and a set E of channels, we formally define an algorithm to be a (W, E)*-stabilizing pulse synchronization protocol with skew* Σ *and accuracy bounds* T^-, T^+ *stabilizing within time* T *with probability* p iff the following holds: Given that during $[t^-, t^+] \supseteq [t^-, t^- + T + \Sigma]$ nodes in W are non-faulty and channels in E correct, with probability at least p a time $t_s \in [t^-, t^- + T]$ exists so that, denoting by $t_i(k)$ the k^{th} time when node i switches to *accept* after t_s ($t_i(k) = \infty$ if no such time exists), for all $i, j \in W$, and $k \in \mathbb{N}$, (i) $t_i(1) \in (t_s, t_s + \Sigma)$, (ii) $|t_i(k) - t_j(k)| \leq \Sigma$ if $\max\{t_i(k), t_j(k)\} \leq t^+$, and (iii) $T^- \leq |t_i(k+1) - t_i(k)| \leq T^+$ if $t_i(k) + T^+ \leq t^+$.

Since the ultimate goal of the pulse generation algorithm is to stabilize a system of DARTS clocks, we introduce an additional port DARTS_i, for each node i, which is driven by node i's DARTS instance. As for other state signals, its output raises flag $\text{Mem}_{i,\text{DARTS}}$, to which for simplicity we refer to as DARTS_i as well. Note that the DARTS signals are of no concern to the liveliness or stabilization of the pulse algorithm itself; rather, they are control signals from the DARTS components that help in adjusting the frequency of pulses to the speed of the DARTS clocks once the system as a whole has become stable. Details can be found in [9].

Basic Cycle. The full algorithm makes use of a rather involved interplay between conditions on timeouts, states, and thresholds to converge to a safe state despite a limited number of faulty components. As our approach is difficult to present in a bulk, we break it down into pieces. Moreover, to facilitate giving intuition about the key ideas of the algorithm, in this section we assume that there are $f < n/3$ faulty nodes, and all the remaining $n - f$ nodes are non-faulty

[8] Note that a single physical fault may cause such a behaviour.

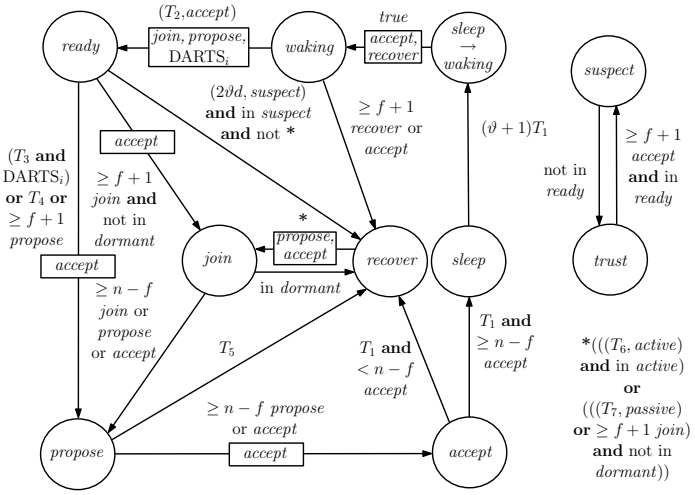

Fig. 1. Overview of the core routine of node i's self-stabilizing pulse algorithm.

within $[0, \infty)$ (where of course the time 0 is unknown to the nodes). We further assume that channels between non-faulty nodes (including loopback channels) are correct within $[0, \infty)$. First, we present the *basic cycle* that is repeated every pulse once a safe configuration is reached. It consists of the states *accept*, *sleep*, *sleep* → *waking*, *waking*, *ready*, and *propose* in the given order (see Fig. 1).

We employ graphical representations of the state machine of each node $i \in V$. States are represented by circles containing their names, while transition $(s, s') \in \mathcal{T}$ is depicted as an arrow from s to s'. The guard $tr(s, s')$ is written as a label next to the arrow, and the reset function's value $re(s, s')$ is depicted in a rectangular box on the arrow. To keep labels simple we make use of abbreviations. We write T instead of (T, s) if s is the state that node i leaves if the condition involving (T, s) is satisfied. Threshold conditions like "$\geq f + 1$ s", where $s \in \mathbb{S}$, abbreviate Boolean predicates that reach over all of node i's memory flags $\text{Mem}_{i,j,s}$, where $j \in V$, and are defined in a straightforward manner. If in such an expression we connect two states by "or", e.g., "$\geq n - f$ s or s'" for $s, s' \in \mathbb{S}$, the summation considers flags of both types s and s'. Thus, it is equivalent to $\sum_{j \in V} \max\{\text{Mem}_{i,j,s}, \text{Mem}_{i,j,s'}\} \geq f + 1$. For any state $s \in \mathbb{S}$, the condition $S_{i,j} = s$, (respectively, $\neg(S_{i,j} = s)$) is written in short as "j in s" (respectively, "j not in s"). If $j = i$, we simply write "(not) in s". We write "true" instead of a condition that is always true. Finally, $re(\cdot, \cdot)$ always requires to reset all memory flags of certain types, hence we write, for example, *propose* if all flags $\text{Mem}_{i,j,propose}$ are to be reset.

We now briefly introduce the basic flow of the algorithm once it stabilizes, i.e., once all $n - f$ non-faulty nodes are well-synchronized. Recall that the remaining up to $f \leq \lfloor n/3 \rfloor$ faulty nodes may produce arbitrary signals on their outgoing channels. A pulse is locally triggered by switching to state *accept*. Thus, assume that at some time all non-faulty nodes switch to state *accept* within a time window of $2d$, i.e., a valid pulse is generated. Supposing that $T_1 \geq 3\vartheta d$, these

nodes will observe, and thus memorize, each other and themselves in state *accept* before T_1 expires. This makes timeout T_1 the critical condition for switching to state *sleep*. From state *sleep*, they will switch to states *sleep* $\rightarrow$ *waking*, *waking*, and finally *ready*, where the timeout $(T_2, \textit{accept})$ is determining the time this takes, as it is considerably larger than $\vartheta(\vartheta+2)T_1$. The intermediate states serve the purpose of achieving stabilization, hence we leave them out for the moment. Note that upon switching to state *ready*, nodes reset their *propose* flags and $\textsc{darts}_i$. Thus, they essentially ignore these signals between the most recent time they switched to *propose* before switching to *accept* and the subsequent time when they switch to *ready*. This ensures that nodes do not take into account outdated information for the decision when to switch to state *propose*. Hence, it is guaranteed that the first node switching from state *ready* to state *propose* again does so because T_4 expired or because T_3 expired and its DARTS memory flag is true. Due to the constraint $\min\{T_3, T_4\} \geq \vartheta(T_2 + 4d)$, we are sure that all non-faulty nodes observe themselves in state *ready* before the first one switches to *propose*. Hence, no node deletes information about nodes that switch to *propose* again after the previous pulse. No non-faulty node can switch to state *accept* before it memorizes at least $n - f$ nodes in state *propose*, as the *accept* flags are reset upon switching to state *waking*. Therefore, at least $n - 2f \geq f + 1$ non-faulty nodes are in state *propose* when the first node switches to *accept* again. Hence, the rule that nodes switch to *propose* if they memorize $f + 1$ nodes in states *propose* will take effect, i.e., the remaining non-faulty nodes in state *ready* switch to *propose* after less than d time. Another d time later all non-faulty nodes in state *propose* will have become aware of this and switch to state *accept* as well, as the threshold of $n - f$ nodes in states *propose* or *accept* is reached. Thus the cycle is complete and the reasoning can be repeated inductively.

Main Algorithm. We proceed by describing the main routine of the pulse algorithm in full. Alongside the main routine, several other state machines run concurrently and provide additional information to be used during recovery.

The main routine is graphically presented in Fig. 1, together with a very simple second component whose sole purpose is to simplify the otherwise overloaded description of the main routine. Except for the states *recover* and *join* and additional resets of memory flags, the main routine is identical to the basic cycle. The purpose of the two additional states is the following: Nodes switch to state *recover* once they detect that something is wrong, that is, non-faulty nodes do not execute the basic cycle as outlined in Section 3. This way, non-faulty nodes will not continue to confuse others by sending for example state signals *propose* or *accept* despite clearly being out-of-sync. There are various consistency checks that nodes perform during each execution of the basic cycle. For example, no non-faulty node may be in state *propose* for more than a certain amount of time before switching to state *accept*. Therefore, nodes will switch from *propose* to *recover* when timeout T_5 expires. Similarly, when in state *ready*, nodes expect others not to be in state *accept* for more than a short period of time, as a non-faulty node switching to *accept* should imply that every non-faulty node switches to *propose* and then to *accept* shortly thereafter. This is expressed by

the second state machine comprising two states only. If a node is in state *ready* and memorizes $f + 1$ nodes in state *accept*, it switches to *suspect*. Subsequently, if it remains in state *ready* until a timeout of $2\vartheta d$ expires, it will switch to state *recover*.

Nodes can join the basic cycle again via the second new state, called *join*. Since the Byzantine nodes may "play nice" towards $f + 1$ or more nodes still executing the basic cycle, making them believe that system operation continues as usual, it must be possible to join the basic cycle again without having a majority of nodes in state *recover*. On the other hand, it is crucial that this happens in a sufficiently well-synchronized manner, as otherwise nodes could drop out again because the various checks of consistency detect an erroneous execution of the basic cycle.

In part, this issue is solved by an additional agreement step. In order to enter the basic cycle again, nodes need to memorize $n - f$ nodes in states *join* (the respective nodes detected an inconsistency), *propose* (these nodes continued to execute the basic cycle), or *accept* (there are executions where nodes reset their *propose* flags because of switching to *join* when other nodes already switched to *accept*). Since there are thresholds of $f + 1$ nodes memorized in state *join* both for leaving state *recover* and switching from *ready* to *join*, all nodes will follow the first one switching from *join* to *propose* quickly, just as with the switch from *propose* to *accept* in an ordinary execution of the basic cycle. However, it is decisive that all nodes are in states that permit to participate in this agreement step in order to guarantee success of this approach.

As a result, still a certain degree of synchronization needs to be established beforehand, both among nodes that still execute the basic cycle and those that do not. For instance, if at the point in time when a majority of nodes and channels become non-faulty, some nodes already memorize nodes in *join* that are not, they may switch to state *join* and subsequently *propose* prematurely, causing others to have inconsistent memory flags as well. Again, Byzantine faults may sustain this amiss configuration of the system indefinitely.

So why did we put so much effort in "shifting" the focus to this part of the algorithm? The key advantage is that nodes outside the basic cycle may take into account less reliable information for stabilization purposes. They may take the risk of metastable upsets (as we know it is impossible to avoid these during the stabilization process, anyway) and make use of randomization. In fact, to make the above scheme work, it is sufficient that all non-faulty nodes agree on a point in time to reset the memory flags for states *join* and *sleep* $\rightarrow$ *waking* as well as certain timeouts, while guaranteeing that no node is in these states close to the respective reset times. Except for state *sleep* $\rightarrow$ *waking*, all of these timeouts, memory flags, etc. are not part of the basic cycle at all, thus nodes may enforce consistent values for them when they agree on such a *resynchronization point*. Conveniently, the use of randomization also ensures that it is quite unlikely that nodes are in state *sleep* $\rightarrow$ *waking* close to a resynchronization point, as the consistency check of having to memorize $n - f$ nodes in state *accept* in order to

switch to state *sleep* guarantees that the time windows during which non-faulty nodes may switch to *sleep* make up a small fraction of all times only.

Consequently, the remaining components of the algorithm deal with agreeing on resynchronization points and utilizing this information in an appropriate way to ensure stabilization of the main routine. We describe this connection to the main routine first. It is done by another, quite simple state machine, which runs in parallel alongside the core routine. It is the machine having three states that is depicted in the upper left corner of Fig. 2.

Its purpose is to reset memory flags in a consistent way and to determine when a node is permitted to switch to *join*. In general, a resynchronization point (locally observed by switching to state *resync*) triggers the reset of the *join* and *sleep* $\rightarrow$ *waking* flags. If there are still nodes executing the basic cycle, a node may become aware of it by observing $f+1$ nodes in state *sleep* $\rightarrow$ *waking* at some time. In this case it switches from state *passive*, which it entered at the point in time when it locally observed the resynchronization point, to state *active*, which enables an earlier transition to state *join*. This is expressed by the rather involved transition rule $tr(\mathit{recover}, \mathit{join})$: T_6 is much smaller than T_7, but T_6 is of no concern until the node switches to state *active* and resets T_6.[9]

It remains to explain how nodes agree on resynchronization points.

Resynchronization Algorithm. The resynchronization routine is specified in Fig. 2 as well. It is a lower layer that the core routine uses for stabilization purposes only. It provides some synchronization that is very similar to that of a pulse, except that such "weak pulses" occur at random times, and may be generated inconsistently after the algorithm as a whole has stabilized. Since the main routine operates independently of the resynchronization routine once the system has stabilized, we can afford the weaker guarantees of the routine: If it succeeds in generating a "good" resynchronization point merely once, the main routine will stabilize deterministically.

Definition 1 (Resynchronization Points). *Given $W \subseteq V$, time t is a W-*resynchronization point *iff each node in W switches to state* supp $\rightarrow$ resync *in the time interval $(t, t+2d)$. A W-resynchronization point is called* good *if no node from W switches to state* sleep *during $(t-(\vartheta+3)T_1, t)$ and no node is in state* join *during $[t-T_1-d, t+4d)$.*

In order to clarify that despite having a linear number of states (supp_i, $i \in V$), this routine can be implemented using constant-bit channels only, we generalize our description of state machines as follows. If a state is depicted as a circle separated into an upper and a lower part, the upper part denotes the local state, while the lower part indicates the signal state to which it is mapped. A node's memory flags then store the respective signal states only, i.e., remote nodes do not distinguish between states that share the same signal.

The basic idea behind the resynchronization algorithm is the following: Every now and then, nodes will try to initiate agreement on a resynchronization point.

[9] The condition "not in *dormant*" ensures that the transition is not performed because of being in state *resync* a long time ago, while there was no recent switch to *resync*.

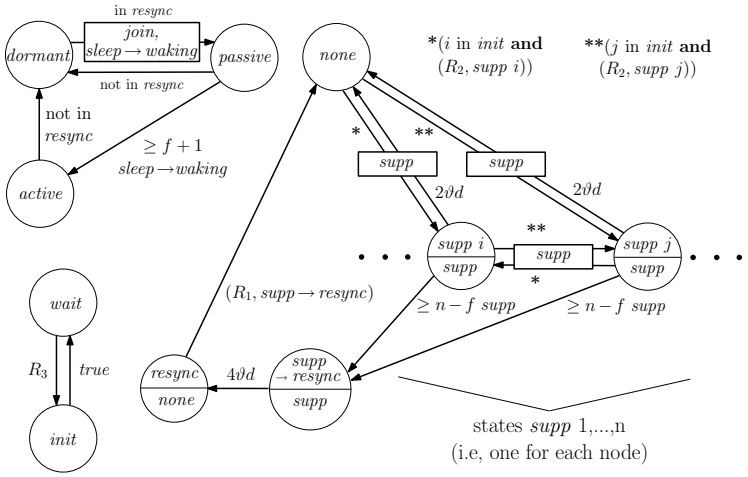

Fig. 2. Resynchronization algorithm of node i.

This is the purpose of the small state machine in the lower left corner of Fig. 2. As the time when a node switches to *init* is determined by the randomized timeout R_3, which we choose to be distributed over a large interval, it is impossible to predict when it will expire, even with full knowledge of the execution up to the current point in time.

Consider now the state machine displayed on the right of Fig. 2. To understand how the routine is intended to work, assume that at the time t when a non-faulty node i switches to state *init*, all non-faulty nodes are not in any of the states *supp* $\rightarrow$ *resync*, *resync*, or *supp* i, and at all non-faulty nodes the timeout $(R_2, supp\ i)$ has expired. Then, no matter what the signals from faulty nodes or on faulty channels are, all non-faulty nodes will be in one of the states *supp* j, $j \in V$, or *supp* $\rightarrow$ *resync* at time $t + d$. Hence, they will observe each other (and themselves) in one of these states at some time smaller than $t + 2d$. These statements follow from the various timeout conditions of at least $2\vartheta d$ and the fact that observing node i in state *init* will make nodes switch to state *supp* i if in *none* or *supp* j, $j \neq i$. Hence, all of them will switch to state *supp* $\rightarrow$ *resync* during $(t, t + 2d)$, i.e., t is a resynchronization point. Since t follows a random distribution that is independent of the remaining algorithm and, as mentioned earlier, most of the times nodes cannot switch to state *sleep* and it is easy to deal with the condition on *join* states, there is a large probability that t is a good resynchronization point. Note that timeout R_1 makes sure that no non-faulty node will switch to *supp* $\rightarrow$ *resync* again anytime soon, leaving sufficient time for the main routine to stabilize.

The scenario we just described relies on the fact that at time t no node is in state *supp* $\rightarrow$ *resync* or state *resync*. We will choose $R_2 \gg R_1$, implying that $R_2 + 3d$ time after a node switched to state *init* all nodes have "forgotten" about this, i.e., $(R_2, supp\ i)$ is expired and they switched back to state *none* (unless other *init* signals interfered). Thus, in the absence of Byzantine faults,

the above requirement is easily achieved with a large probability by choosing R_3 as a uniform distribution over some interval $[R_2 + 3d, R_2 + \Theta(nR_1)]$: Other nodes will switch to *init* $\mathcal{O}(n)$ times during this interval, each time "blocking" other nodes for at most $\mathcal{O}(R_1)$ time. If the random choice picks any other point in time during this interval, a resynchronization point occurs. Even if the clock speed of the clock driving R_3 is manipulated in a worst-case manner (affecting the density of the probability distribution with respect to real time by a factor of at most ϑ), we can just increase the size of the interval to account for this.

However, what happens if only *some* of the nodes receive an *init* signal due to faulty channels or nodes? If the same holds for some of the subsequent *supp* signals, it might happen that only a fraction of the nodes reaches the threshold for switching to state *supp* $\rightarrow$ *resync*, resulting in an inconsistent reset of flags and timeouts across the system. Until the respective nodes switch to state *none* again, they will not support a resynchronization point again, i.e., about R_1 time is "lost". This issue is the reason for the agreement step and the timeouts $(R_2, \textit{supp } j)$. In order for any node to switch to state *supp* $\rightarrow$ *resync*, there must be at least $n - 2f \geq f + 1$ non-faulty nodes supporting this. Hence, all of these nodes recently switched to a state *supp* j for some $j \in V$, resetting $(R_2, \textit{supp } j)$. Until these timeouts expire, $f + 1 \in \Omega(n)$ non-faulty nodes will ignore *init* signals on the respective channels. Since there are $\mathcal{O}(n^2)$ channels, it is possible to choose $R_2 \in \mathcal{O}(nR_1)$ such that this may happen at most $\mathcal{O}(n)$ times in $\mathcal{O}(n)$ time. Playing with constants, we can pick $R_3 \in \mathcal{O}(n)$ maintaining that still a constant fraction of the times are "good" in the sense that R_3 expiring at a non-faulty node will result in a good resynchronization point.

Analysis and Results. Due to lack of space, we had to relegate the detailed formalization and analysis of our algorithm to [9]. We will hence only briefly summarize our major results, along with the required constraints. With $\lambda := \sqrt{(25\vartheta - 9)/(25\vartheta)} \in (4/5, 1)$, we need:

$$
\begin{array}{ll}
T_1 \geq \vartheta 4d & T_2 \geq (3\vartheta + 1 - 1/\vartheta)T_1 + T_5 \\
T_3 \geq (2\vartheta^2 + 3\vartheta - 1)T_1 - T_2 + \vartheta(T_6 + 5d) & T_4 \geq T_3 \\
T_5 \geq (\vartheta^2 + \vartheta - 2)T_1 + \vartheta(T_2 + T_4 + 9d) - T_6 & T_6 \geq \vartheta((\vartheta + 1)T_1 + T_2 + 6d) \\
T_7 \geq (\vartheta - 1)T_2 + \vartheta((1 + 2/\vartheta - \vartheta)T_1 + T_4 + T_5 + T_6 + 10d) & R_1 \geq \vartheta(T_7 + (4\vartheta + 8)d) \\
R_2 \geq 2\vartheta(R_1 + (\vartheta + 2)T_1 + T_2/\vartheta + (8\vartheta + 9)d)(n - f)/(1 - \lambda) & R_3 = \text{uniformly distributed on} \\
\vartheta\lambda \leq (T_2 - (4\vartheta^3 + 28\vartheta^2 + 4\vartheta)d)/(T_2 - (8\vartheta^2 + \vartheta)d) & \quad [\vartheta(R_2 + 3d), \vartheta(R_2 + 3d) + 8(1 - \lambda)R_2]
\end{array}
$$

This system is solvable for any $\vartheta < \vartheta_{\max} \approx 1.247$ with $T_1, \ldots, T_7, R_1 \in \mathcal{O}(1)$ and $R_2 \in \mathcal{O}(n)$. Furthermore, the system must satisfy the following property during a given time interval $[t^-, t^+]$: There is a subset $W \subseteq V$ of size at least $n - \lfloor n/3 - 1 \rfloor$ such that during $[t^- - (\vartheta(R_2 + 3d) + 8(1 - \lambda)R_2) - d, t^+]$ (i) all nodes $i \in W$ are non-faulty, and (ii) all channels $S_{i,j}$, $i, j \in W$, are correct.

A safe configuration is reached once all nodes in W switch to *accept* within $3d$: Time t is a *stabilization point (quasi-stabilization point)* iff all nodes $i \in W$ switch to *accept* within $[t, t + 2d)$ ($[t, t + 3d)$).

Theorem 1. *Suppose t is a quasi-stabilization point. Then (i) all nodes in W switch to* accept *exactly once within $[t, t+3d)$, and (ii) there will be a stabilization point $t' \in (t+(T_2+T_3)/\vartheta, t+T_2+T_4+5d)$ satisfying that no node in W switches to* accept *in the time interval $[t+3d, t')$ and that (iii) each node i's, $i \in W$, states*

*of the basic cycle (*accept*,* sleep*,* sleep → waking*,* waking*,* ready*, and* propose*) are metastability-free during* $[t+3d, t'+4d)$.

By induction, we see that if for $|W| \geq n - \lfloor n/3 - 1 \rfloor$ we can show the existence of a quasi-stabilization point $t \in [t^-, t^+]$, then for any $E \supseteq W^2$ the protocol is (W, E)-stabilizing with skew $2d$ and accuracy bounds $(T_2 + T_3)/\vartheta - 2d$ and $T_2 + T_4 + 7d$. As it is impossible to guarantee metastability-freedom during stabilization, our remaining statements argue about metastability-free executions only, i.e., provided that metastability does not occur, the system will stabilize.

Theorem 2. *Denote by* $\hat{E}_3 := \vartheta(R_2 + 3d) + 8(1-\lambda)R_2 + d$. *For any* $k \in \mathbb{N}$ *and any time* $t \in [t^-, t^+ - (k+1)\hat{E}_3]$, *with probability at least* $1 - (1/2)^{k(n-f)}$ *there will be a good* W*-resynchronization point in* $[t, t + (k+1)\hat{E}_3]$.

A good resynchronization point ensures sufficient consistency of nodes' memories for the main routine (Fig. 1) to stabilize deterministically.

Theorem 3. *Let* $T_k := (k+2)\hat{E}_3 + R_1/\vartheta$ *for* $k \in \mathbb{N}$. *Then, with probability at least* $1 - 1/2^{k(n-f)}$, *a stabilization point in* $[t^-, t^- + T_k]$ *exists, and the algorithm stabilizes within time* T_k *plus longest timeout.*

References

1. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: Proc. 27th symposium on Principles of Distributed Computing (PODC). pp. 385–394 (2008)
2. Berman, A., Keidar, I.: Low-Overhead Error Detection for Networks-on-Chip. In: The 27th International Conference on Computer Design (ICCD) (2009)
3. Bhamidipati, R., Zaidi, A., Makineni, S., Low, K., Chen, R., Liu, K.Y., Dalgrehn, J.: Challenges and Methodologies for Implementing High-Performance Network Processors. Intel Technology Journal 6(3), 83–92 (2002)
4. Chapiro, D.M.: Globally-Asynchronous Locally-Synchronous Systems. Ph.D. thesis, Stanford University (1984)
5. Constantinescu, C.: Trends and Challenges in VLSI Circuit Reliability. IEEE Micro 23(4), 14–19 (2003)
6. Daliot, A., Dolev, D.: Self-Stabilizing Byzantine Pulse Synchronization. CoRR abs/cs/0608092 (2006)
7. Daliot, A., Dolev, D., Parnas, H.: Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks. In: Proc. 6th Symposium on Self-Stabilizing Systems (SSS) (2003)
8. Dike, C., Burton, E.: Miller and Noise Effects in a Synchronizing Flip-Flop. IEEE Journal of Solid-State Circuits SC-34(6), 849–855 (1999)
9. Dolev, D., Függer, M., Lenzen, C., Schmid, U.: **F**ault-tolerant **A**lgorithms for **T**ick-Generation in **A**synchronous **L**ogic: Robust Pulse Generation. CoRR abs/cs/1105.4708 (2011)
10. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM 51(5), 780–799 (2004)
11. Friedman, E.G.: Clock Distribution Networks in Synchronous Digital Integrated Circuits. Proceedings of the IEEE 89(5), 665–692 (2001)

12. Fuchs, G., Függer, M., Steininger, A.: On the Threat of Metastability in an Asynchronous Fault-Tolerant Clock Generation Scheme. In: Proc. 15th Symposium on Asynchronous Circuits and Systems (ASYNC). pp. 127–136. Chapel Hill, N. Carolina, USA (2009)
13. Függer, M.: Analysis of On-Chip Fault-Tolerant Distributed Algorithms. Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik (2010)
14. Függer, M., Dielacher, A., Schmid, U.: How to Speed-Up Fault-Tolerant Clock Generation in VLSI Systems-on-Chip via Pipelining. In: Proc. 8th European Dependable Computing Conference (EDCC). pp. 230–239 (2010)
15. Függer, M., Schmid, U.: Reconciling Fault-Tolerant Distributed Computing and Systems-on-Chip. Research Report 13/2010, Technische Universität Wien, Institut für Technische Informatik (2010)
16. Függer, M., Schmid, U., Fuchs, G., Kempf, G.: Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In: Proc. 6th European Dependable Computing Conference (EDCC). pp. 87–96 (2006)
17. Gadlage, M.J., Eaton, P.H., Benedetto, J.M., Carts, M., Zhu, V., Turflinger, T.L.: Digital Device Error Rate Trends in Advanced CMOS Technologies. IEEE Transactions on Nuclear Science 53(6), 3466–3471 (2006)
18. Hoch, E., Dolev, D., Daliot, A.: Self-stabilizing Byzantine Digital Clock Synchronization. In: Proc. 8th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006). vol. 4280, pp. 350–362 (2006)
19. Internat. Technology Roadmap for Semiconductors (2007), http://www.itrs.net
20. Kinniment, D.J., Bystrov, A., Yakovlev, A.V.: Synchronization Circuit Performance. IEEE Journal of Solid-State Circuits SC-37(2), 202–209 (2002)
21. Malekpour, M.: A Byzantine-Fault Tolerant Self-stabilizing Protocol for Distributed Clock Synchronization Systems. In: Proc. 9th Conference on Stabilization, Safety, and Security of Distributed Systems (SSS). pp. 411–427 (2006)
22. Marino, L.: General Theory of Metastable Operation. IEEE Transactions on Computers C-30(2), 107–115 (1981)
23. Metra, C., Francescantonio, S., Mak, T.: Implications of Clock Distribution Faults and Issues with Screening them During Manufacturing Testing. IEEE Transactions on Computers 53(5), 531–546 (2004)
24. Pease, M., Shostak, R., Lamport, L.: Reaching Agreement in the Presence of Faults. Journal of the ACM 27, 228–234 (1980)
25. Polzer, T., Handl, T., Steininger, A.: A Metastability-Free Multi-synchronous Communication Scheme for SoCs. In: Proc. 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009). pp. 578–592 (2009)
26. Portmann, C.L., Meng, T.H.Y.: Supply Noise and CMOS Synchronization Errors. IEEE Journal of Solid-State Circuits SC-30(9), 1015–1017 (1995)
27. Restle, P.J., others;: A Clock Distribution Network for Microprocessors. IEEE Journal of Solid-State Circuits 36(5), 792–799 (2001)
28. Semiat, Y., Ginosar, R.: Timing Measurements of Synchronization Circuits. In: Proc. 9th Symposium on Asynchronous Circuits and Systems (ASYNC) (2003)
29. Srikanth, T.K., Toueg, S.: Optimal Clock Synchronization. Journal of the ACM 34(3), 626–645 (1987)
30. Sundaresan, K., Allen, P., Ayazi, F.: Process and temperature compensation in a 7-MHz CMOS clock oscillator. IEEE J. Solid-State Circuits 41(2), 433–442 (2006)
31. Teehan, P., Greenstreet, M., Lemieux, G.: A Survey and Taxonomy of GALS Design Styles. IEEE Design and Test of Computers 24(5), 418–428 (2007)
32. Widder, J., Schmid, U.: The Theta-Model: Achieving Synchrony without Clocks. Distributed Computing 22(1), 29–47 (2009)